NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

ANALYSIS OF TLCHARTS FOR WEAPON SYSTEMS SOFTWARE DEVELOPMENT

by

Kadir Alpaslan Demir

December 2005

Thesis Advisor:	Doron Drusinsky
Thesis Co-Advisor:	Man-Tak Shing

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** December 2005	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Analysis of TLCharts for Weapon Systems Software Development			**5. FUNDING NUMBERS**
6. AUTHOR(S) Kadir Alpaslan Demir			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

The success of formal specifications and reactive systems is highly dependant on the formal specification language being used. To date, the most common approach to this problem involves two activities: (i) the specification activity, where correctness properties are specified, and (ii) verification activity, where the system under review is proven to satisfy those properties. Typically, some form of temporal logic or regular expression language is used to specify the correctness properties; properties that are specified for given states of the system under review. This means that specification is partial and is done after system design, prototyping, or coding. Temporal logics have been found to be unsuitable for early specification.

This thesis investigates the suitability of TLCharts, a specification language that combines statecharts and temporal logic, for the early specification of the dynamic characteristics of a homing torpedo. In order to achieve the task, a fictitious homing torpedo example, called KTorp, is used. Using a systematic approach, we developed deterministic statecharts and non-deterministic TLCharts for the KTorp control software. Our case study shows that using TLCharts as the early specification language for weapon systems software provides efficient, visual and intuitive specifications.

14. SUBJECT TERMS Weapon Systems Software Development, Temporal Logic, State Charts, TLCharts, Homing Torpedo Software			**15. NUMBER OF PAGES** 107
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

ANALYSIS OF TLCHARTS FOR WEAPON SYSTEMS SOFTWARE DEVELOPMENT

Kadir Alpaslan Demir
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1999

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

and

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 2005

Author: Kadir Alpaslan Demir

Approved by: Doron Drusinsky
Thesis Advisor

Man-Tak Shing
Thesis Co-Advisor

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The success of formal specifications and reactive systems is highly dependant on the formal specification language being used. To date, the most common approach to this problem involves two activities: (i) the specification activity, where correctness properties are specified, and (ii) verification activity, where the system under review is proven to satisfy those properties. Typically, some form of temporal logic or regular expression language is used to specify the correctness properties; properties that are specified for given states of the system under review. This means that specification is partial and is done after system design, prototyping, or coding. Temporal logics have been found to be unsuitable for early specification.

This thesis investigates the suitability of TLCharts, a specification language that combines statecharts and temporal logic, for the early specification of the dynamic characteristics of a homing torpedo. In order to achieve the task, a fictitious homing torpedo example, called KTorp, is used. Using a systematic approach, we developed deterministic statecharts and non-deterministic TLCharts for the KTorp control software. Our case study shows that using TLCharts as the early specification language for weapon systems software provides efficient, visual and intuitive specifications.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION....1

II. WEAPON SYSTEMS SOFTWARE DEVELOPMENT....3
- A. SOFTWARE PROCESS....3
  - 1. Concept Exploration Phase....4
  - 2. Requirements Analysis Phase....4
  - 3. Specification Phase....5
  - 4. Design Phase....5
  - 5. Implementation Phase....6
  - 6. Integration Phase....6
  - 7. Maintenance Phase....7
  - 8. Retirement....7
  - 9. Testing....8
  - 10. Documentation....8
- B. CAPABILITY MATURITY MODELS....9
- C. SOFTWARE LIFE-CYCLE MODELS....11
  - 1. Waterfall Model....12
  - 2. Incremental Model....13
  - 3. Spiral Model....15
- D. CHALLENGES OF WEAPONS SYSTEMS SOFTWARE DEVELOPMENT....16
  - 1. Definition of Weapons Systems Software....17
  - 2. General Characteristics of Weapons Systems Software Development....18
    - *a. Technical Characteristics of Weapon Systems Software Development....18*
    - *b. Managerial Characteristics of Weapon Systems Software Development....20*
  - 3. Previous Studies on Military Software....21
  - 4. Technical Challenges of Weapons Systems Software Development....22

III. TLCHARTS....27
- A. HAREL STATECHARTS....27
  - 1. Introduction....27
  - 2. State-levels: Clustering and Refinement....28
  - 3. Orthogonality: Independence and Concurrency....30
  - 4. Conditions, Actions and Activities....32
  - 5. Condition and Selection Entrances....33
  - 6. Delays and Timeouts....35
  - 7. Drawbacks of Harel Statecharts....35
- B. TEMPORAL LOGIC....36
  - 1. Classical Logic....37

2. Modal Logic 37
3. Temporal Logic 38
4. Examples of Temporal Logics 40
C. TLCHARTS 42
1. An Infusion Pump Keypad Control Example 44
IV. A CASE STUDY: KTORP 49
A. KTORP INTRODUCTION 49
B. KTORP TECHNICAL SPECIFICATIONS 50
C. SAFETY-RELATED SPECIFICATIONS 51
D. DYNAMIC BEHAVIOR OF KTORP 51
1. Enable Phase 51
2. Search Phase 52
3. Attack Phase 52
E. SEARCH MODES 52
1. Snake Search 53
2. Circular Search 55
F. HIGH LEVEL USE CASES 57
1. Snake Search Use Case 58
2. Circular Search Use Case 60
G. SYSTEM SEQUENCE DIAGRAMS 62
H. STATECHARTS 65
I. ASSERTIONS 78
V. CONCLUSION AND FUTURE WORK 83
A. SUMMARY 83
B. FUTURE WORK 85
LIST OF REFERENCES 87
INITIAL DISTRIBUTION LIST 91

LIST OF FIGURES

Figure II-1. Five Levels of SW-CMM 11
Figure II-2. Derivation of the Original Model Proposed by Royce [After 9] 13
Figure II-3. Representation of the Incremental Model [After 4]....14
Figure II-4. Full Spiral Model [From 10]....16
Figure II-5. Code Size/Complexity Growth [From 11]....24
Figure III-1. Simple State Diagram 28
Figure III-2. Introduction of Superstate E 29
Figure III-3. Top-Down Approach of the System 29
Figure III-4. Uses of Entering-By-History, Initial State, and Final State....30
Figure III-5. Statechart of the Car System....31
Figure III-6. Examples of Actions and Conditions....32
Figure III-7. Examples of Activities....33
Figure III-8. Example of a Conditional 34
Figure III-9. Example of a Selection [From 2] 34
Figure III-10. Examples of a Delay and a Timeout 35
Figure III-11. Example of a Race Condition 36
Figure III-12. Relations between TLCharts and its Constituents (Syntax) 43
Figure III-13. Deterministic Harel Statechart Specification for Requirement R1 [From 37] 46
Figure III-14. TLChart Specification for Requirement R1 [From 37] 47
Figure IV-1. Illustration of KTorp Sections 50
Figure IV-2. Snake Search Pattern Part 1 54
Figure IV-3. Snake Search Pattern Part 2 54
Figure IV-4. Circular Search Pattern Part 1....56
Figure IV-5. Circular Search Pattern Part 2....56
Figure IV-6. Use Case Diagram of KTorp....58
Figure IV-7. System Sequence Diagram for KTorp Snake Search 63
Figure IV-8. System Sequence Diagram for KTorp Circular Search 64
Figure IV-9. Highest Level Statechart Specification of KTorp....66
Figure IV-10. Transition between “Enable_Phase” State and “Search_Phase” State 67
Figure IV-11. Refinement of “Enable_Phase” Composite State for Snake Search Mode....68
Figure IV-12. Refinement of “Enable_Phase” Composite State for Circular Search Mode ..68
Figure IV-13. Transition between “Enable_Phase” State and “Search_Phase” State 69
Figure IV-14. Transition from “Search_Phase” State to “Attack_Phase” State....70
Figure IV-15. Refinement of “Attack_Phase” Composite State for Snake Search Mode....71
Figure IV-16. An Attempt to Achieve “Attack_Phase” Composite State for Snake Search Mode 72
Figure IV-17. Sliding Window Problem....73
Figure IV-18. Deterministic Harel Statechart Specification for the Requirement SS.6 74
Figure IV-19. Non-deterministic Solution for Requirement SS.6 75
Figure IV-20. Primary Statechart Specification for Attack Phase....77
Figure IV-21. Highest Level TLChart Specification of KTorp with a Temporal Logic Conditioned Transition 80

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table II-1. System Functionality Requiring Software [From 11].....17
Table III-1. Comparing Features of Temporal Logics [From 27].....42

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGEMENTS

This thesis is a result of a long study. It investigates a subject using a new specification language. It required continuous learning throughout the study. I hope this is just the beginning of better studies.

First, I have to thank to my beautiful wife who supported me for a long time. She continuously sacrificed many things for my achievement. She was beside me during my long studies. I could not have achieved without her. Thank you, Zeynep. Thank you.

Also, thanks to my thesis professors, they helped and waited for the final product for a long time.

Finally, I want to thank my family, my friends, my professors and everyone who helped me reach this day.

To my wife, Zeynep…

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Science helps mankind to understand the world and to ease human lives. It is an undeniable fact that the biggest supporters of science were kings, lords, bureaucrats, or in short, governments. One of the most important motivational factors for such support was to acquire better weapon systems and it worked. Scientists developed new and better weapons, and governments used these weapons in wars, defense structures and dominance strategies. It is fair to say that this quest also helps humanity in different aspects of life. Some examples from recent history include the development of nuclear weapons, the development of computers (i.e., ENIAC) for flight calculations in missile projectiles, and so forth. The development of nuclear weapons made it possible to benefit from nuclear energy while the development of computers and transistors created benefits arising from the substantial automation occurring in almost every field of human practices. Therefore, this thesis, states that such a quest to develop better weapon systems software will also be beneficial to other systems software.

Today, a regrettable belief about computers is that computer systems break frequently. This view may be true for daily life, but for safety-critical systems, and especially for weapon systems, such failures are completely unacceptable. Generally, the failure of a safety-critical system will endanger a human life or lives. Most weapon systems are in the category of safety-critical systems. Thus, the development of weapon systems is an expensive and thorough process, which poses many challenges. This thesis will address one aspect of the development process challenges: the capturing of the system requirements with complete correctness, without omissions, and the easing of the verification of specifications using a formal specification language.

The goal of this thesis is to analyze a formal visual specification language called "TLCharts" [1], proposed by Drusinsky, in the context of weapon systems software development. TLCharts is a hybrid of Harel Statecharts [2] and temporal logic. The thesis includes a case study to compare the results with a well-known and widely-used visual specification language, Statecharts. This case study is a homing torpedo, named "KTorp".

Conclusions drawn from developing the torpedo system in both methods and comparing the correctness, expressiveness, and effectiveness of capturing the system behavior will finalize the study.

The weapon systems development process will be explained in Chapter II. This chapter will also address the challenges in the development and will briefly explain the methods used to ease the challenges.

As mentioned earlier, TLCharts is a hybrid of Harel Statecharts and temporal logic. Statecharts and temporal logic are both a type of formal specification language. Chapters III and IV will provide a background on temporal logic and Statecharts. This background will help to analyze and compare TLCharts with these specification languages.

Chapter V will define KTorp, a fictitious homing torpedo system. The requirements for the system will be derived from the system definition and use cases. System sequence diagrams derived from use cases will be included. Using the use cases and system sequence diagrams, the statechart of the homing torpedo system will be developed. Then, the system will be specified using TLCharts.

An assessment of the study and the conclusions will conclude this thesis.

II. WEAPON SYSTEMS SOFTWARE DEVELOPMENT

The main activities related to software development are the same regardless of the type of software. However, the type of software affects the rigorousness of the phases within the development. Some systems software such as safety-critical real-time reactive systems software development pose many challenges [3] and most weapon systems software are actually this type. In addition to these challenges, weapon systems software incorporates others. The goal of this chapter is to identify these differences from a commercial software development process.

First, the main activities of the software development process will be explained briefly. Next, widely-accepted software life-cycle models will be introduced. Finally, the chapter will end with the identified differences between weapon systems software development and other software development. It is important to note that there is little available published work on the subject of weapon systems software development. Therefore, this chapter introduces the subject and identifies the need for more detailed analysis which may be a prospective thesis subject.

A. SOFTWARE PROCESS

The software process is the way to produce software. It incorporates all the activities related to software life cycle as well as the tools used and the individuals building the software [4]. Regardless of the software type, the software process involves eight main phases: concept exploration, requirements analysis, specification, design, implementation, integration, maintenance, and retirement. Activities within these phases and transitions from one phase to another are defined with software life-cycle models and vary within models. The readers familiar with the software development process may argue that some of the main activities are missing from the list: testing and documentation. In fact, these activities are not missing; they are already incorporated into every phase. These activities will be discussed separately.

Considering the software separate from a system or disregarding the underlying hardware will lead to a false approach in systems context. Thus, it is important to keep in mind that the phases explained here must be thought of as system development phases.

1. Concept Exploration Phase

Many software engineering textbooks do not address concept exploration as a separate phase. However, the activities related to this phase require special attention. Some textbooks mention this phase within requirements analysis and this approach is partly true because some activities are overlapping. Nonetheless, some activities such as developing a technology that will make the system more effective or cheaper are not directly related to requirements.

The concept exploration phase involves activities related to the investigation of risky areas within the development. In this phase, key enabling technologies and features are researched, analyzed, and even validated in some developments. Developers responsible for various phases brainstorm about the concepts and processes that will bring the system to life. Strategies and approaches are identified to attack risky areas. Precautions are acknowledged to meet the budget and schedule. Also in some cases, the tools and techniques to be used in the development process are considered. A good example of this supportive task is observed within the development of the Ballistic Missile Defense System (BMDS).

This phase is crucial for complex systems, which is usually the case for weapon systems. The reason is that complex systems may present some hidden challenges and often these challenges are not obvious initially.

2. Requirements Analysis Phase

In this phase, the developers identify the requirements of the system. Mainly, the requirements are extracted from stakeholders. A stakeholder is an individual, an entity, or a regulation that has expectations from a system in one way or another. For a missile system, stakeholders are the branch of military service, the government, the personnel who will use the system, the military standards, or the environment. Some requirements do not derive from stakeholders but are enforced by the way the software is produced or what is currently state of the art.

In short, requirements analysis is the agreement between the developer and the stakeholders about what the system should accomplish. It is the developers' responsibility to differentiate what the customer wants and needs and even renegotiate the requirements.

Requirements analysis is the most important phase of any software development, because the most costly errors are introduced during this phase. Fixing a requirement error drastically increases in later phases.

3. Specification Phase

After the customer and the developer agree on what the system should do, the specification document is prepared. The methods used in the requirements phase are generally informal or semi-formal. In the specification phase, the methods are more formal and the outputs should be as clear as possible. In this phase, the functionalities of the product are explicitly identified, and documented. The specifications may be in natural language. For weapon systems software, in most cases, they are also formalized. The formalization is in the form of some specification language. There are many specification languages. Examples include Z, VDM, Temporal Logic, Statecharts, and SPEC. [2, 27, 39, 40]

There are many automated tools associated with formal specification languages. Many of these tools incorporate automatic syntactic checking and documentation aids. Therefore, using a formal specification language allows developers to verify that there are no inconsistent declarations in the specification. However, semantic checks of specifications still require human analysis.

Formal or semi-formal specifications should not be ambiguous, incomplete, and contradictory. Also, specifications must be traceable to requirements. Formal specification helps transition to the design phase from the requirements phase in a methodological manner.

4. Design Phase

The specification identifies *what* the system does while the design clarifies *how* the system performs [4]. Using the specification as an input, the design team creates a design. The designers decompose the system into modules. This decomposition is

sometimes called architectural design. Then, the design team works on each module, and identifies the details of the modules. The final product of this work is called detailed design. These designs form an input for the implementation.

Design reviews are an important part of this phase. In these reviews, the design is checked according to the specifications.

5. Implementation Phase

Coding of system modules is the main activity of this phase. The coding activity should be fairly simple, if the previous phases are well achieved. Sometimes the choice of programming language may become an issue in this phase, but this selection is better done in the concept exploration phase. Every programming language is designed for some purpose. Several may not be suitable for some projects. The right tool for the right job leads to success. For example, the programming language of choice for a large portion of weapons system software was Ada and still is. Ada possesses several built in features such as strong-type checking, which enforces good programming practices.

After enough modules are implemented, the developers begin to integrate the system. Code reviews are found to be useful in many successful projects.

6. Integration Phase

The integration phase is the phase in which the system modules are combined to form the system. In theory, if the previous phases are quite successful, this phase should be straightforward. Generally, however, this is not the case. A large number of errors surface in the integration phase. Fixing some of these errors even requires returning to the requirements phase, which may affect a significant portion of the product. Weapon systems are usually complex systems. The integration of complex systems poses many challenges and the solution to these challenges lies in the good business practices during the previous phases.

After integration, the system is deployed. Deployment is also another difficult task for many weapon systems. The main reason behind the difficulty is that it is very hard to create the environment of the system in a laboratory setting, which limits the testing of the system under real conditions.

7. Maintenance Phase

After the system is deployed and accepted by the customer, all the changes made to the system are considered maintenance phase activities. A significant portion of software life cycle costs is devoted to the maintenance phase. This portion can reach up to 80% for many organizations [5]. It is obvious that the longer the life cycle, the higher the maintenance costs. The life cycle of the weapon systems is generally longer compared to commercial systems, which increases the importance of maintenance activities for weapon systems software.

There are three types of maintenance: corrective, perfective, and adaptive. Fixing faults after development due to specification faults, design faults or any other type of faults is called corrective maintenance. Increasing the effectiveness of software is called perfective maintenance. Sometimes, the environment of the software product changes; therefore the product may have to be ported to a new compiler, a new operating system or new hardware. This type of maintenance is called adaptive maintenance.

The importance of software maintenance within software life cycle management is unquestionable. However, to improve software maintenance activities, earlier phases of software development must be improved.

8. Retirement

After years of service, the system will come to a point that maintenance is no longer cost-effective. Some reasons are:

- The environment of the system is so different that modifications are too expensive.
- The proposed changes may require a major design effort that is too costly.
- The documentation is not updated as necessary; therefore maintenance is becoming too costly.
- The system is no longer useful and incapable of satisfying the current mission needs.

Retirement of weapon systems may be an important issue, which is different than many commercial systems. For example, some weapon systems use radioactive materials and the disposal of these materials necessitates cautious consideration. Retirement may require careful planning, which should be done in steps. The plan should consider the effects of removing components on the systems safety.

9. Testing

It is important to note that software testing has a special definition:

> Testing is the process of executing a program/system with the intent of finding errors. [6]

In the context of this thesis, verification, validation, and software testing activities are discussed under the title of testing. Therefore, it is not incorrect to stress that each phase should incorporate some of the activities under this title.

In many software life cycle models, testing is a separate phase, which is conducted after the implementation or integration phases. This approach has many drawbacks. Software development is an expensive activity and the discovery of errors late in the process will be costly. Therefore, testing should be conducted at any point possible. In the requirements analysis phase, the requirements should be validated with the customers. In the specifications phase, the specifications should be verified and validated according to requirements and similar activities should be conducted for the rest of the software process. Every effort to find an error in any phase prior to advancement to next phase is valuable and definitely a cost-effective effort.

10. Documentation

The importance of documentation is almost unquestionable. For example, one of the main problems with the maintenance phase is inadequate documentation or a lack of documentation. It was mentioned earlier that software maintenance cost can reach up to 80% of the software life-cycle cost. As a result, documentation considerably affects the cost of the software life cycle. However, the pressure of delivering the product on time and within budget often negatively affects documentation efforts. Thus, documentation becomes a secondary priority whereas it should have the same priority as the delivery of a success product.

One of the main goals of computer-aided software development is implicit support for documentation. Nevertheless, the software process is far from having full automation.

Documentation significantly affects weapon systems software development more. The reason lies in the necessity of conforming to the standards set by the government. This has quality considerations as well as cost considerations. The details will be discussed later in the thesis.

B. CAPABILITY MATURITY MODELS

The software crisis has attracted a lot of attention in the last three decades. Incomplete projects, cost and schedule overruns, and unsuccessful systems have become a major concern due to the increased use of software intensive systems in every aspect of life. In 1987, U.S. Department of Defense reported:

> After two decades of largely unfulfilled promises about productivity and quality gains from applying new software methodologies and technologies, industry and government organizations are realizing that their fundamental problem is the inability to manage the software process. [7]

DoD founded the Software Engineering Institute (SEI) due to related issues in December 1984. The institute resides at Carnegie Mellon University. One of the most important contributions of SEI is the Capability Maturity Model (CMM). There is also another international software process improvement effort by International Standards Organization, the ISO/IEC 15504.

The capability maturity models include a variety of models:

- CCM for software (SW-CMM)
- CCM for management of human resources (P-CMM)
- CCM for systems engineering (SE-CMM)
- CCM for integrated product development (IPD-CMM)
- CCM for software acquisition (SA-CMM)

In 1997, it was decided to integrate some of these models into one common framework under the name capability maturity model integration (CCMI) [4]. SW-CMM focuses on improving the management of the software process. The intent is that a better product and better techniques will result from improving software process management. CMM identifies five *maturity* levels. The higher the maturity level, the better the development process. [8]

- **Maturity Level 1. Initial Level:** Being the lowest, this level indicates that no sound software engineering practices are in place. The software process is completely ad hoc and success is very much dependent on the existing staff.
- **Maturity Level 2. Repeatable Level:** This level indicates that basic software management practices are in place. The experiences in similar projects are repeatable. Managers are proactive in determining problems and taking corrective actions.
- **Maturity Level 3. Defined Level:** This level indicates that software process is fully documented. Technical and managerial aspects of projects are clearly defined. Reviews for quality assurance are in place and the software process is analyzable. Therefore, the software process is open to improvements.
- **Maturity Level 4. Managed Level:** Levels 4 and 5 are only reached by very few organizations. In level 4, organizations have clear goals about quality and productivity. These goals are monitored continually. Quantitative measurements guide the improvements for quality and productivity goals. The software process is managed.
- **Maturity Level 5. Optimizing Level:** This level is the highest level in SW-CMM. The goal at this level is continuous improvement of the software process. The experience gained in each project is analyzed and utilized for future projects. Level 5 incorporates this positive feedback loop to optimize the software process.

Figure II-1 shows the five levels of SW-CMM taken from Software Process Self-Assessment Training Participant's Guide, Software Engineering Institute, Carnegie Mellon University, 1989.

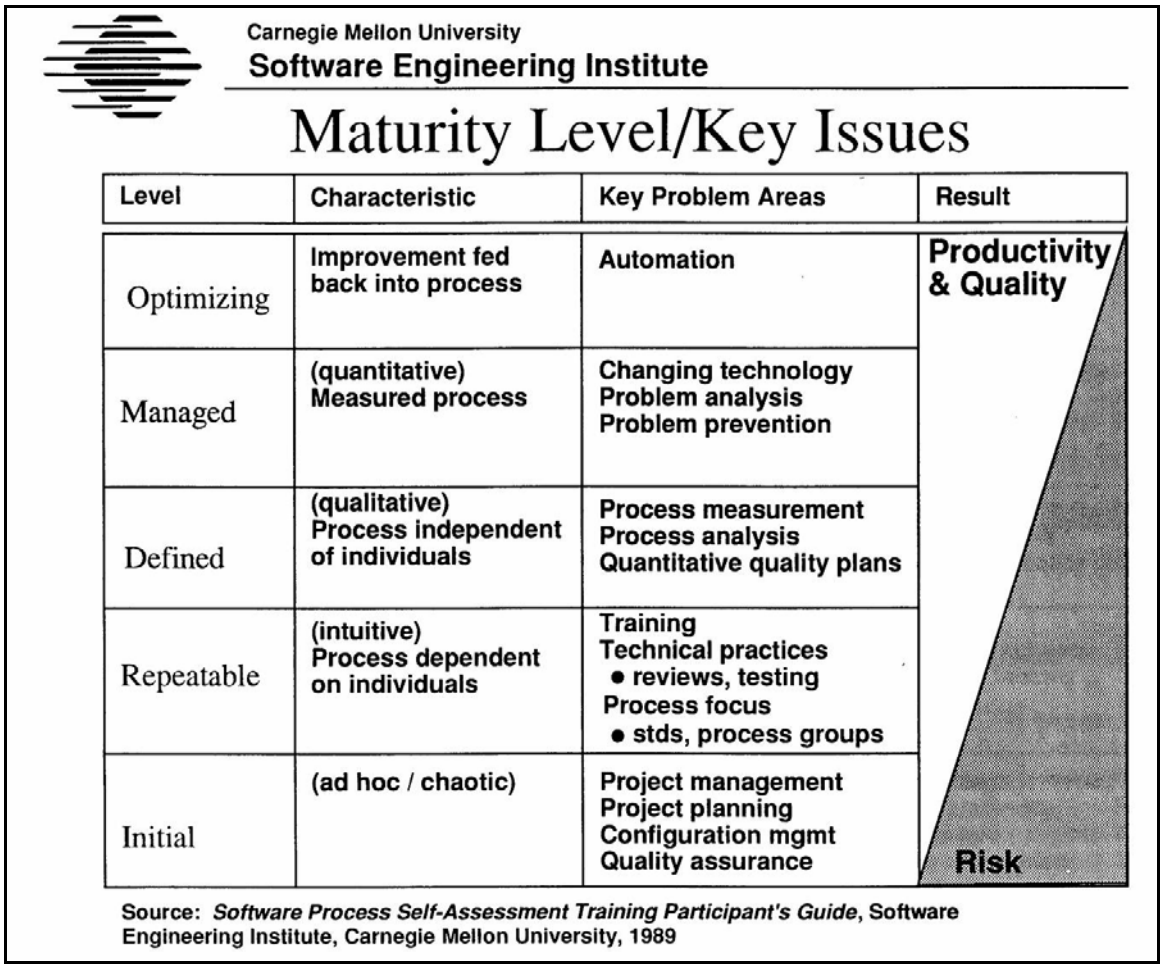


Figure II-1. Five Levels of SW-CMM

C. SOFTWARE LIFE-CYCLE MODELS

The series of steps for each software product as it progresses is called a software life-cycle model [4]. Basically, a software product begins as a conceptual idea. Next, the requirements for the realization of the product are identified. The specifications and the design for the product are accomplished, forming an input for the implementation and integration. Deployment and maintenance are the following activities in the life of the product. The life cycle ends with retirement. Each software life-cycle model defines the necessary steps, the order and the interaction of the steps within the life cycle to accomplish the project successfully.

The selection of the life-cycle model depends on various factors. The main factors are the type of software and the practices used by the organization. Many software life-cycle models were proposed in the past but only a few were used effectively and gained

attention. Only the most commonly used models for weapon systems software development will be explained in this thesis.

1. Waterfall Model

The waterfall model was the only widely accepted model until early 1980's [4]. It was first recognized by W.W. Royce [9]. There are many variations of this basic model addressing some of the weaknesses. This model has been successfully used in many projects. It is important to note that using this model is not applicable to every type of software intensive system.

The model basically starts with the identification of system requirements. These higher requirement analyses lead to software requirements. The analysis follows the software requirements. This phase can be mapped to the specification phase in a modern day model but it incorporates additional activities such as risk analysis. Program design is the phase in which the software architecture and the detailed design is accomplished. Coding, or in other words implementation, follows this phase. In the first versions of the model, integration is not a clear phase but it is an activity within coding. Testing of the software is the next step in the life-cycle model. Finally, the original model ends with the operations phase. This phase includes maintenance and retirement in a modern day model.

Royce identifies two important concepts for the life-cycle model. The first one is the feedback with the previous phase. According to him, every phase must have a feedback loop with its preceding step. This feedback reduces the probability of errors passing into next step. The second important concept Royce emphasizes is the importance of documentation. He advised that all documentation should be completed before advancing to the next step in the life-cycle.

In the original model, Royce introduced another step that is not commonly used today. That step is called "Preliminary Program Design" and includes designing a database and processors, a documenting system overview, allocating subroutine storage and subroutine execution times as well as describing operating procedures [9]. If the time period is considered, most of the technologies in place today were not used at that time.

Therefore, such an analysis called preliminary program design was a necessity. Today, with the use of advanced tools and techniques, most of these activities do not require a separate phase.

A derivation of the original model proposed by Royce [9] is shown in Figure II-2. This figure is very close to Royce's original model. Today, the names of the phases are modified and the model is known as the "Waterfall Life-cycle Model".

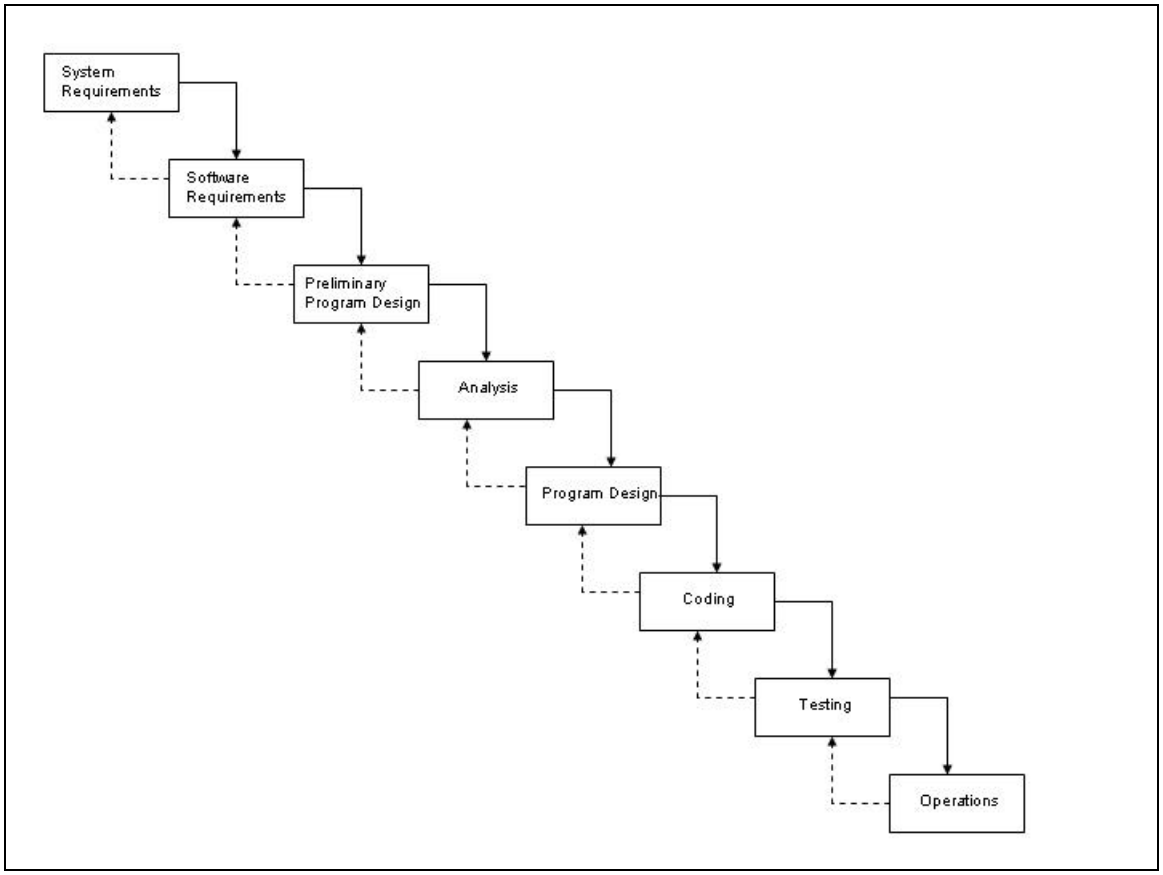


Figure II-2. Derivation of the Original Model Proposed by Royce [After 9]

2. Incremental Model

Software is a product that is built step by step. This process is often called the incremental process. This aspect of the software is captured by a life-cycle model called the "Incremental Model". In this model, the software is built by a series of incremental *builds*. Every build captures one or more features of the product. Then, the build is added and the system is tested as a whole.

In the incremental model, the requirements are extracted at first. Afterwards, the system specification is prepared. An architectural design that will create the system is accomplished. Next, the developers separate the system into features satisfying a number of requirements. To implement one or more features is the goal of each build. The builds may be accomplished concurrently or one at a time depending on a specific system. Each build is tested and integrated into the system. Afterwards, the system is tested. This process continues until the product is finished.

In this model, there is an operational-quality product at every stage, which satisfies a subset of the client's requirements. A representation of an incremental model is given in Figure II-3 [4].

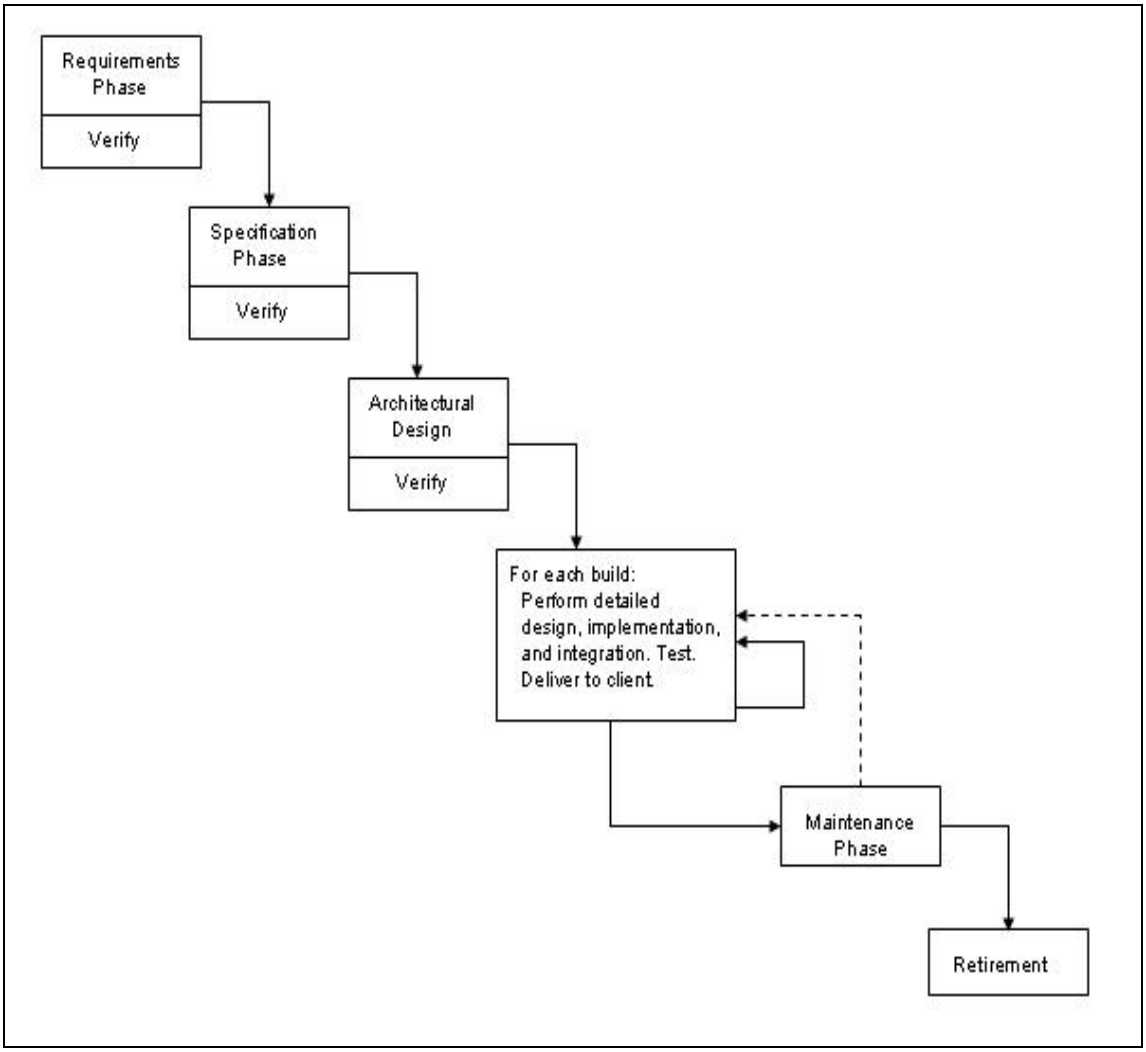


Figure II-3. Representation of the Incremental Model [After 4]

3. Spiral Model

The spiral model [10] was proposed by B. W. Boehm in May 1988. Since that time, it has been widely recognized and used successfully. This model addresses one of the main issues observed in developing a complex large-scale system, which is risk. Basically in a spiral model, every phase starts with a risk analysis. If the risks are found to be unsolvable, then the project is canceled. Actually, this is only the case for in-house projects, but the risk analysis activity is useful for productivity. Therefore, the spiral model is also applicable to contracted projects. Another main issue addressed by the model is the difficulty of requirements extraction. Prototyping is the proposed solution to this problem in the spiral life-cycle model.

The spiral model can be thought of as a waterfall model with the inclusion of risk analysis and prototyping at every phase. A representation of the spiral model is given in Figure II-4. In the figure, the radial dimension represents cumulative cost to date, and the angular dimension represents progress through the spiral. Each phase is represented by a spiral. Each phase starts with the left upper quadrant by determining the objectives, alternatives to those objectives, and constraints for alternatives. This activity results with a strategy. Afterwards, this strategy is analyzed with the associated risks. These risks are resolved and the phase enters into the right lower quadrant. This part of the spiral is actually the same as the waterfall model. Finally, the phase ends with planning the next phase.

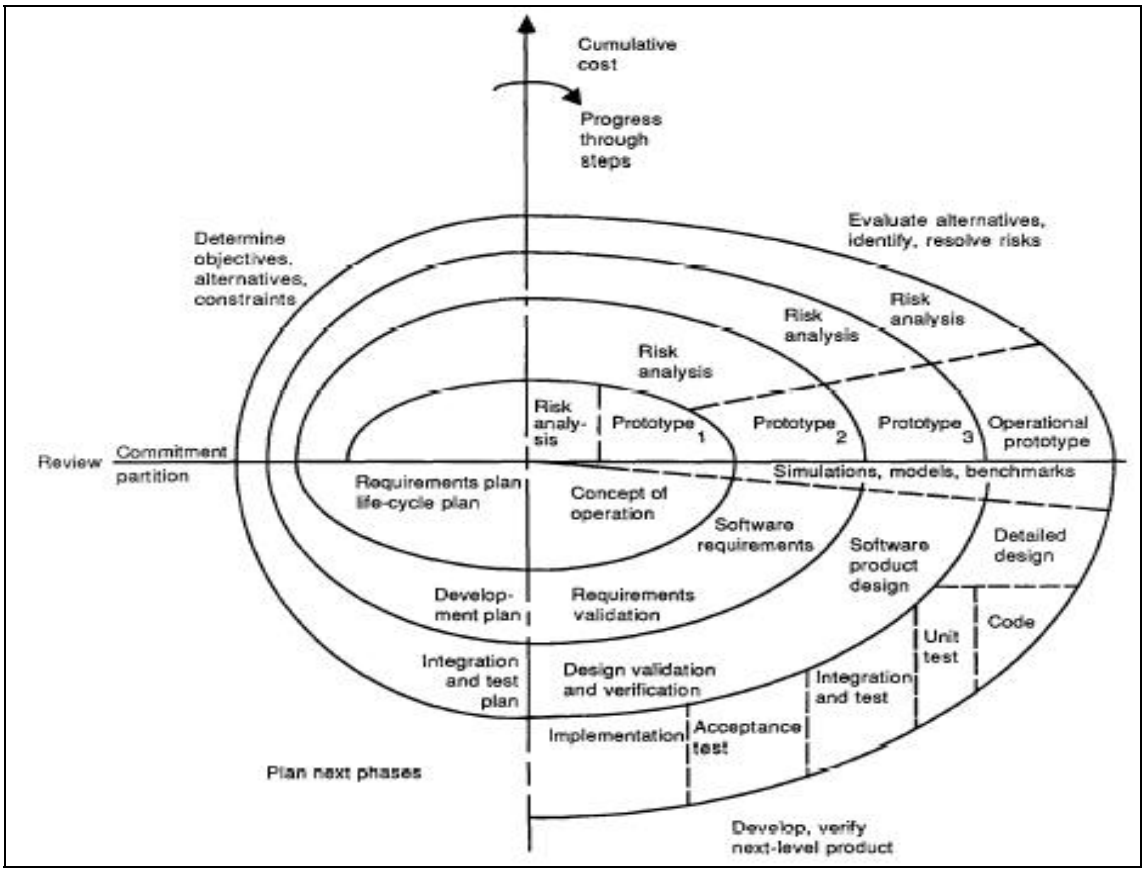


Figure II-4. Full Spiral Model [From 10]

The spiral model especially addresses the issues related to large-scale systems. It is mostly suitable for weapon systems software development due to the high risks associated with this type of project.

D. CHALLENGES OF WEAPONS SYSTEMS SOFTWARE DEVELOPMENT

It was previously mentioned that a significant portion of weapons systems are safety-critical real-time reactive systems. In today's environment, these systems rely more and more on software-based components. Table II-1 shows the system functionality requiring software for a typical weapon system, which is combat aircraft.

Weapon System	Year	% of Functions Performed in Software
F-4	1960	8
A-7	1964	10
F-111	1970	20
F-15	1975	35
F-16	1982	45
B-2	1990	65
F-22	2000	80
Source: PM Magazine		

Table II-1. System Functionality Requiring Software [From 11]

This table simply shows the increasing importance of the software portion in a weapon systems development. The Crosstalk–Journal of Defense Software Engineering published an article entitled "Now More Than Ever, Software Is the Heart of Our Weapons Systems" in its January 2002 issue. This article provides examples of software intensive systems that increase the U.S.'s national defense capabilities. It is fair to say that the success of the weapon system is becoming merely dependent upon the success of the software portion of the system. Therefore, this part of the thesis will analyze the challenges posed by military software and mostly will focus on the technical differences between military and commercial software.

It is important to note that the software itself is not different whether it is military or commercial. However, the software process is not about the software only. It is about the manner in which the software is produced. Thus, the software process is seriously affected by the managerial aspects and these aspects influence the technical portion of software development.

1. Definition of Weapons Systems Software

The term weapon systems software is in fact self-explanatory. However, in literature, this term is not commonly used. Instead, the terms "Military Software" or "Defense Software" is generally used to encapsulate weapons systems software. The phrase "Defense Software" refers to software produced for a uniformed military service [19]. The term also includes software produced for the Department of Defense, or the

equivalent in other countries. An article by Capers Jones [19] provides a broad definition of defense software and the main attribute that distinguishes defense software from other types of software:

> The broad definition of defense software includes a number of subclasses such as software associated with weapons systems; with command, control, and communication systems (usually shortened to C3 or C cubed); with logistical applications; and also with software virtually identical to civilian counterparts such as payroll applications, benefits tracking applications, and the like. The main attribute that distinguishes defense software from other types of software is adherence to military or DoD standards.

In the context of this thesis, the term "Military Software" and "Defense Software" will be used interchangeably. When the term "Weapon Systems Software" is used, it will refer to software associated with weapons systems. Examples of these weapon systems include missiles, torpedoes, artillery systems, combat aircrafts, and fire control systems.

2. General Characteristics of Weapons Systems Software Development

The general characteristics of weapons systems software development is categorized under two titles in this thesis: technical and managerial characteristics. The technical characteristics refer to the issues related to the product itself and challenges imposed by these types of software. Managerial characteristics are inherent to the weapon systems software development environment. Weapons systems are built in-house by government branches or contracted to companies by governments. System acquisition in this context has significant overhead if compared to commercial systems acquisition. These issues are addressed under managerial characteristics.

a. Technical Characteristics of Weapon Systems Software Development

At present, it is very hard to find a piece of weapon system software that is not safety-critical real-time reactive software. Also, many weapon systems are complex systems. Some of the main characteristics of weapon systems are as follows:

- Real-Time Systems: Real-time systems are concurrent systems with timing constraints [16]. A real-time system generally consists of sensors, actuators, and real-time system core. Weapon systems are real-time systems. For example, a missile consists of sensors that capture signals from targets, actuators in wings that will help to navigate, and a decision-making component which can be thought of as the real-time system core.

Meeting the timing constraints between these components poses significant challenges. Also, real-time systems require real-time control that is making control decisions based on input data, without any human intervention [16]. The development of such systems requires at least the use of formal methods, a well-developed architecture, an extensive design and testing efforts.

- Safety-Critical Systems: A safety-critical system is a system whose failure may cause injury or death to human beings. Actually, weapon systems are intended to cause the destruction on targets. However, the key phrase in a weapon systems context is "to intended targets". Therefore, weapon systems should not cause harm to its own users. For example, currently a torpedo is incapable of differentiating signals from a target or from the mother ship. Therefore, developers incorporate a feature in the design such arming after a safe distance. The failure of this feature may cause the torpedo to attack its mother ship. Ensuring the correct implementation of safety-criticality to weapon systems is another challenging part of weapon systems software development. Safety-critical aspects of systems are also shared by some commercial applications such as medical systems.
- Mission-Critical Systems: Mission-critical systems are systems whose failure will cause significant loss in terms of money, trust, or defense capabilities of a nation or of a military entity. For example, if a particular weapon does not work in a combat airplane or in a ship, the airplane or the ship will be subject to destruction by the enemy. The development of mission-critical systems poses similar challenges as in the case of real-time systems.
- Embedded Systems: Embedded systems are systems that are parts of a larger hardware/software system. These systems often require special purpose hardware entailing increased optimization. The software associated with these systems has to work with this hardware and requires more attention. Signal processing components (that may be hardware, software or firmware-based) found in weapon systems are examples of such systems.
- Interaction with External Environment: Real-time systems typically interact with an external environment that generally does not involve humans [16]. In this environment, the interactions are generally not periodic; therefore timing constraints can be more complex. For example, a combat system may include a close-in weapons system to protect the ship from incoming missiles. This requires monitoring the signals in the environment and initiating the necessary components of various weapon systems components.
- Reactive Systems: Reactive systems are the systems that respond to external stimuli in a limited time period. Many real-time systems are reactive systems [17]. This is also the case for weapon systems. These types of systems are event-driven and must respond to external stimuli

[16]. In most cases, the system has to keep a history of previous events to determine the corresponding response. For example, in a torpedo system, the number of captured signals from a target within a specific time period may initiate a number of actions. The correct implementation of reactive systems poses different challenges than many other systems due to the unpredictable nature of their environments.

- High Quality Systems: Weapon systems must be high quality systems. Current state of the art does not have a precise quality metric for software systems. However, quality includes attributes such as reliability, performance, fault-tolerance, safety, security, availability, testability, and maintainability [18]. Some of these attributes have metrics, but the software engineering discipline is far from having a quality metric. The quality in weapons system software is understood by user satisfaction, which is mostly after the fact. Generally, weapon systems must be reliable, available, safe, maintainable, and fault-tolerant. In short, they have to include almost all quality attributes. Software Productivity Research Incorporation has been measuring software quality and productivity since 1985. Their findings indicate that defense systems projects rank at the top in software quality [19]. However, defense systems also rank last in terms of software productivity. Developing high quality software is a labor intensive task and requires software development best practices.

b. Managerial Characteristics of Weapon Systems Software Development

Weapon systems software acquisition and development is different than commercial systems acquisition. The main reason behind this difference is that having the government as a customer enforces serious overhead in the process. Capers Jones reports in his article [19] that defense software projects rank last in terms of software productivity. The main reason is that Department of Defense standards created a number of extra tasks for defense software that do not occur in the civilian sector, which is also the same for other countries.

The report of the Defense Science Board Task Force on Military Software [13] is merely focused on the difficulties of military software development. The report identifies the following in its executive summary section.

> In spite of the substantial technical development needed in requirements-setting, metrics, and measures, tools, etc., the Task Force is convinced that today's major problems with military software development are not technical problems, but management problems.

This finding is also supported by a more recent report (November 2000) by the Defense Science Board.

The major challenge of defense systems including weapon systems is due to acquisition policies. In acquisition, the government releases a general requirements documentation. Potential contractors develop specifications based on this documentation. Then, the government grants the projects on a better value bidding policy. After the project is granted, the requirements become inflexible and almost freeze in many cases. This is contradictory to the nature of software development. Even if the contractor wants to change some of the initial requirements, the process is so cumbersome that the change becomes risky in the development cycle.

Another main challenge imposed by the government is that defense software requires must be compatible with many standards and regulations. Another consequence of this requirement is the extensive documentation to show such compatibilities.

The details of managerial characteristics of weapon systems development is already addressed in other studies and these studies are listed in the previous studies in military section of the thesis.

3. Previous Studies on Military Software

Previous studies on military software are limited. The major reports and publications on the issue are produced by the Defense Science Board [12] established in 1956 under the Assistant Secretary of Defense. This board was created to prepare reports related to military software from time to time. Some of the reports are as follows:

- Report of the Defense Science Board Task Force on Military Software – 1987
- Defense Science Board Task Force on Acquiring Defense Software Commercially – 1994
- Report of the Defense Science Board Tasks on Open Systems – 1998
- Report of the Defense Science Board Task Force On Defense Software – 2000

There are also other reports prepared by the science panels of the military services and the National Research Council:

- Adapting Software Development Policies to Modern Technology-1989 [14]
- The Report of the AMC Software Task Force -1989
- Scaling Up: A Research Agenda For Software Engineering – Computer Science and Technology Board Research Council- 1989

Crosstalk – The Journal of Defense Software Engineering is also another source of information. This journal is an approved Department of Defense journal. In every issue, the journal addresses specific issues related to defense software. [15] However, there is no specific issue addressing this topic.

Also, none of these studies or articles includes a clear-cut analysis of weapon systems software. The necessity of such differentiation is still up for debate. However, this thesis will not address this issue.

4. Technical Challenges of Weapons Systems Software Development

Addressing all the challenges of weapons systems development is not the purpose of this study. However, it is an issue that many researchers have been studying for years. Thus, in this part of this thesis, only a few of the issues, especially those different from commercial software development, will be addressed.

The report of the Defense Science Board Task Force on Defense Software [11] generally focuses on the managerial aspects of the development. However, the report also briefly mentions the technical challenges.

The United States is the major producer and consumer of defense software in the world and by away the leader [19]. Most of the researches on related issues are also conducted in this country. The findings of this research are also applicable to other countries.

Most of the issues identified in this study are the projections of managerial aspects of weapons systems development. There are also inherent challenges due to the type of software produced. These inherent challenges are also seen in commercial software development.

Some of the technical challenges of weapons systems development are as follows:

- The applicability of the waterfall model to software projects is limited. This model only works well for custom-developed software where

requirements are fixed when the design begins. Most of the software projects still employ a waterfall model [11]. This, in fact, is a side effect of the current software acquisition policies.

- One of the enforcements of the waterfall model is the extensive documentation required after each phase. This, in fact, works well with the acquisition policy. Documentation is also very important for maintenance. However, the documentation itself requires maintenance just as does the software. Whenever a change occurs in the software, the documentation also needs to be modified as necessary. If the documentation is more than adequate, then modification is an extra cost. How much documentation is enough and necessary is openly debated for weapons systems software development.
- Having the government as a customer forces the contractors to comply with many standards, regulations, and policies. This, in fact, helps the customers to develop high quality systems. However, compliance issues drive up the cost and reduce productivity. The volume of defense software specifications and other paper documents has been three times larger when compared to the civilian sector [19]. Also, the software engineering discipline is a fast-emerging discipline. The standards must keep pace with current practices, and this issue is burdensome for government agencies.
- The programming language Ada was enforced by the DoD in many projects for nearly two decades. Using Ada as the programming language in military software was a recommendation in the 1987 report[13]. Since then, the use of Ada became a choice in many weapons systems software development. The benefits of using Ada are also recognized by the commercial sector, especially in aviation systems. On the other hand, new programming languages have been introduced. One of them is Java, which offers a platform independent development environment. This property made the Java programming language a widely-used and productive environment. However, Ada has always found limited usage, which restricts the number of language experts. The choice of Ada in weapons systems software development has come into question due to the newest technologies introduced everyday.
- Lacking a quality metric for software poses another challenge for weapons systems software development in which the quality of the product is unquestionable crucial. According to a study conduced by the Standish Group [20], only 16% of all IT projects are completed on time and on budget. The study includes a variety of projects from commercial and defense software. Having a quality metric will play a key role in increasing this percentage. Although, the software engineering discipline is far from creating such a metric.
- Most of the weapons systems software development requires longer schedules when compared to similar developments in the commercial world. Just reaching a final decision on military software contracts results

in a delay of between six to 18 months [19] and this is even before the work starts. When the schedule is long, the requirements are more likely to change. In a weapons systems context, software will work with specialized hardware and the hardware technology advances very rapidly. Therefore, even the change in hardware technology will necessitate requirements changes in software.

- The requirements extraction in the weapons systems software process is particularly harder than many other commercial applications. The main reason behind this challenge is that there are many shareholders in weapons systems software. The list starts with the government, the service, the standards, and the users. Even some of the weapons systems are developed with the participation of many countries. The F-16 combat airplane and Harpoon missile are examples of such weapons. This is hardly the case for many commercial systems.
- Defense software is often more complex than commercial software. This is a consequence of requirements to provide greater functionality and higher reliability than commercial systems [11]. Figure II-.5 shows code size over complexity for various systems. The development of complex systems is an obvious challenge. Mostly military software must integrate with many other systems and legacy systems.

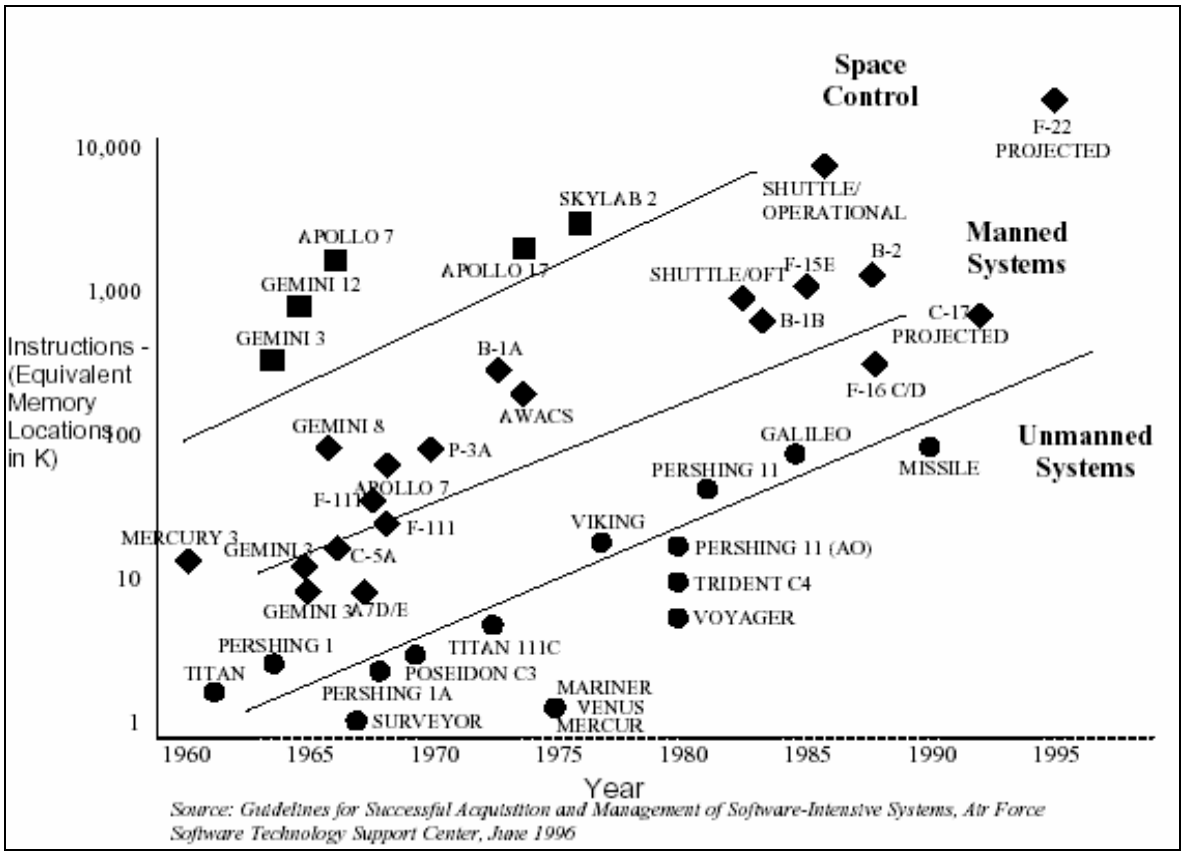


Figure II-5. Code Size/Complexity Growth [From 11]

- Information security is an important issue for governments. Weapons systems software development is also affected by this issue. The development of these weapons must occur in a secure environment. Also, the product must be resistible to malicious attacks. In other words, the weapons systems software must be secure. The development of secure applications has been a hot topic for many years.

THIS PAGE INTENTIONALLY LEFT BLANK

III. TLCHARTS

As mentioned in the introduction, TLCharts [1], proposed by Drusinsky, is a hybrid of Harel Statecharts [2], and temporal logic. It combines the power of the two specification languages. This chapter will provide a brief introduction to Harel Statecharts, temporal logic, and TLCharts.

A. HAREL STATECHARTS

1. Introduction

Statecharts is a widely accepted specification language. The main reason for its acceptance is that statecharts are capable of representing the behavioral aspects of reactive systems in a visual form. A reactive system, unlike a transformational system, is generally event-driven. It continuously reacts to external and internal stimuli. The behavior of a reactive system can be characterized as the set of allowed sequences of input and output events, conditions, and actions.

Finite state machines and their corresponding state diagrams have been used to formally describe sequential logic circuits for a long time. State diagrams are directed graphs, showing the states with nodes and the transitions with arrows. The arrows are labeled with the triggering events and guarding condition. However, state diagrams are incapable of describing complex systems due to the exponentially growing number of states and transitions. They are also flat, which eliminates the capability of a top-down formalism. In order to successfully represent complex reactive systems, a state/event approach must be modular, hierarchical and well-structured [2]. Statecharts were proposed as a solution to overcome such challenges.

Since statecharts were first defined by Harel [2], many variants of the language were proposed in the literature. A survey in 1994 discusses nearly 20 variants [21]. Statecharts is an unofficial language. Therefore, no complete specification of the language exists. Today, two of the variants are used extensively. The first is Harel Statecharts and the second is UML Statecharts [22]. These two variants and many other variants are supported by the tools. Thus, the semantics of the language is dependent on the tools that the developers use.

This thesis will briefly introduce Harel statecharts as proposed in [2]. The detailed description of the language can be found in [2, 17, 23]. All the variants including UML Statecharts are extended and modified versions of Harel Statecharts.

Harel, briefly explains the main aspects of the language in [2] as

> Statecharts constitute a visual formalism for describing states and transitions in a modular fashion, enabling clustering, orthogonality (i.e., concurrency) and refinement, and encouraging 'zoom' capabilities for moving easily back and forth between levels of abstraction.
>
> statecharts = state-diagrams + depth + orthogonality + broadcast-communication.

2. State-levels: Clustering and Refinement

The most important drawback of the use of state diagrams in a complex systems representation is that state diagrams are flat. Therefore, statecharts overcome this problem by means of clustering and refinements.

Figure III-1 shows a simple state diagram of a specification.

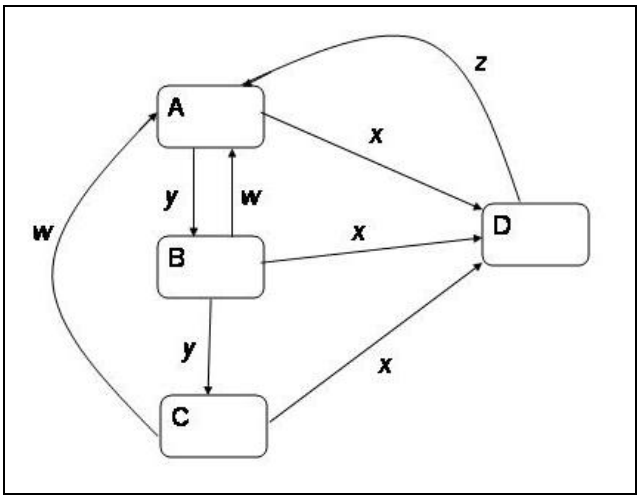


Figure III-1. Simple State Diagram

In the state diagram, the states are represented with rounded rectangles. The names of the states are shown on the upper right corner of the states. Transitions between states are represented with arrows. The labels on the arrows are the events that cause state transitions. In the state diagram, event *x* takes the system to state D from states A, B, or

C. Therefore, these states can be clustered to one superstate as shown in Figure III-2. The new superstate is called state E.

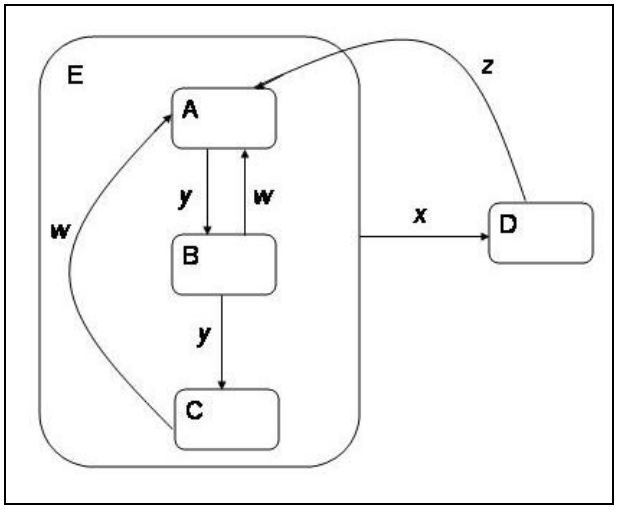


Figure III-2. Introduction of Superstate E

If Figures III-1 and III-2 are compared, note that the number of transitions is reduced by two in this simple state diagram. The semantics of E is the exclusive-or (XOR) of state A, B, and C. When the system is in superstate E, actually the system can only be in state A, B, or C. Therefore, this superstate is an abstraction of the states.

The system can also be analyzed by a top-down approach. Consider Figure III-3.

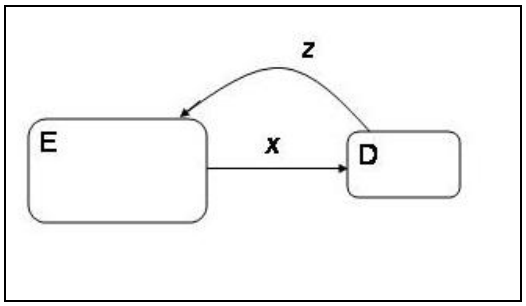


Figure III-3. Top-Down Approach of the System

Figure III-3 shows the same system without the details of state E. Using a top-down approach, state E will be refined and the system state diagram will become the diagram shown in Figure III-2.

The example shows the use of clustering and refinement. Clustering or abstraction is a bottom-up concept. Refinement is a top-down approach. These two mechanisms enable the representation of complex systems in a feasible way.

Sometimes, the system may require remembering the last state visited in a group of states when entering that group. This is called "entering-by-history" [2]. The history within a superstate is denoted by the letter H surrounded by a circle. The history can be applied to the lowest level in the hierarchy by attaching an asterisk to the H.

The initial state of a statechart is represented by an arc originating from a small black circle. The final state is denoted by a larger black circle inside a bigger white circle.

The use of entering-by-history, initial state, and final state is shown in Figure III-4.

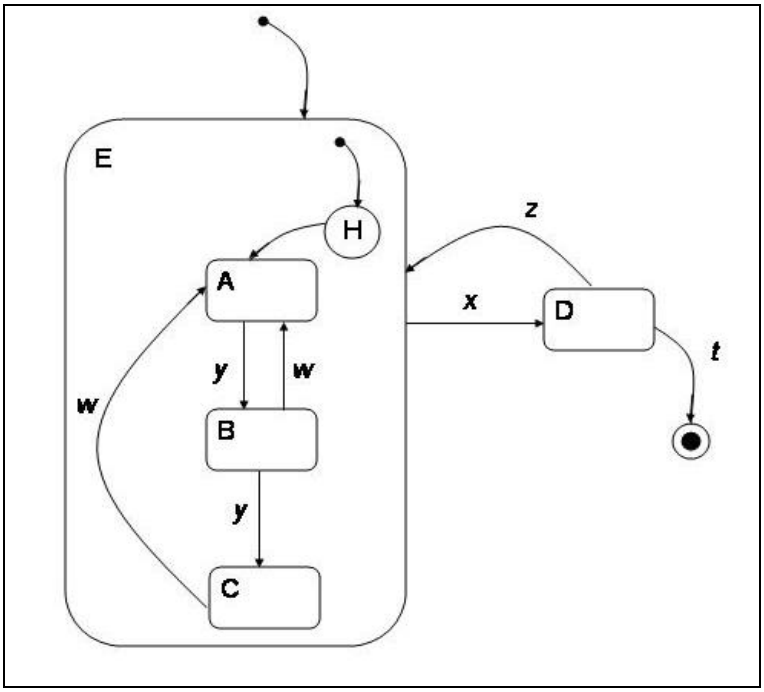


Figure III-4. Uses of Entering-By-History, Initial State, and Final State

3. Orthogonality: Independence and Concurrency

The orthogonality of statecharts explains AND decomposition of states. When the system must be in all of its AND components, the orthogonality property of statecharts is used. Consider a simple example. The motor of a car is either running or stopping and the radio of the car may be on or off. Assume the system is the car. The state of the car is

denoted by state A. State B represents the motor state, and the state C represents the radio state. Figure III-5 shows the statechart of the car system.

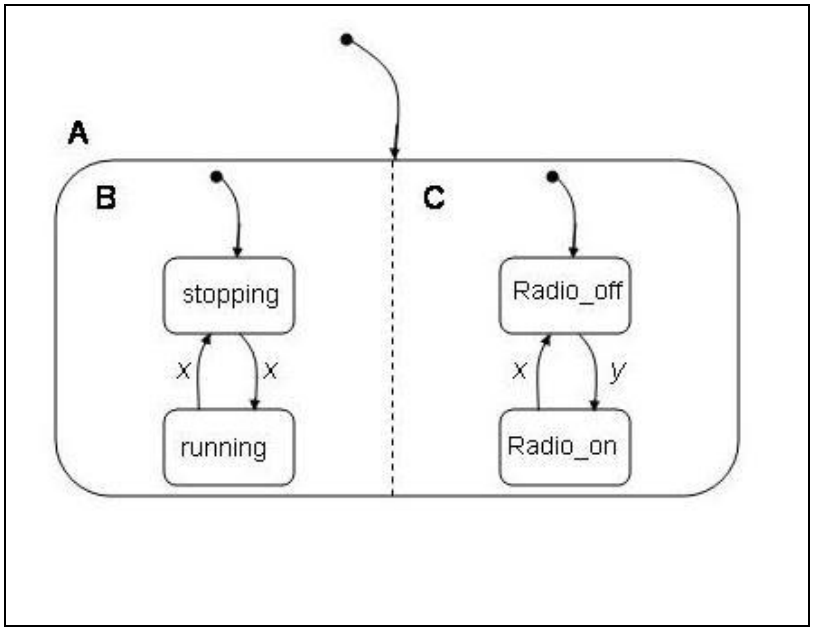


Figure III-5. Statechart of the Car System

When the car is in the default state, it is stopping and its radio is off. The event *x* denotes the event of turning the car key. After the driver turns the key, the car starts running. The event *y* represents the event of pushing the radio on button. While the car is running, the driver turns on the radio by event *y*. Then, the driver wants to stop the car and he turns the key. This event causes two transitions in the diagram. The first one causes the car to stop and the second causes the radio to be off (if it is on).

Figure III-5 illustrates state A consisting of AND components B and C. Therefore, state A is the orthogonal product of states B and C. Event *x* is the synchronization event which takes the system to the stopping and radio_off states at the same time. The independence property of statecharts is shown by event *y*; since this event only takes the system from the radio_off state to the radio_on state. Both behaviors are part of orthogonality of states B and C, which is used to describe the AND decomposition [2]. The lack of orthogonality in finite state machines causes the formalism to suffer from an exponentially increasing number of states.

4. Conditions, Actions and Activities

In statecharts, transitions can be labeled in the form e*(C)/A,* where e is the event triggering the transition, C is the condition, and A is the action associated with the transition. Conditions are also known as guards. In this form, a transition only occurs if the event is triggered and the condition is true. Actions are carried out by the system and they take zero time ideally. Events and actions are closely related; actions may be events for other transitions.

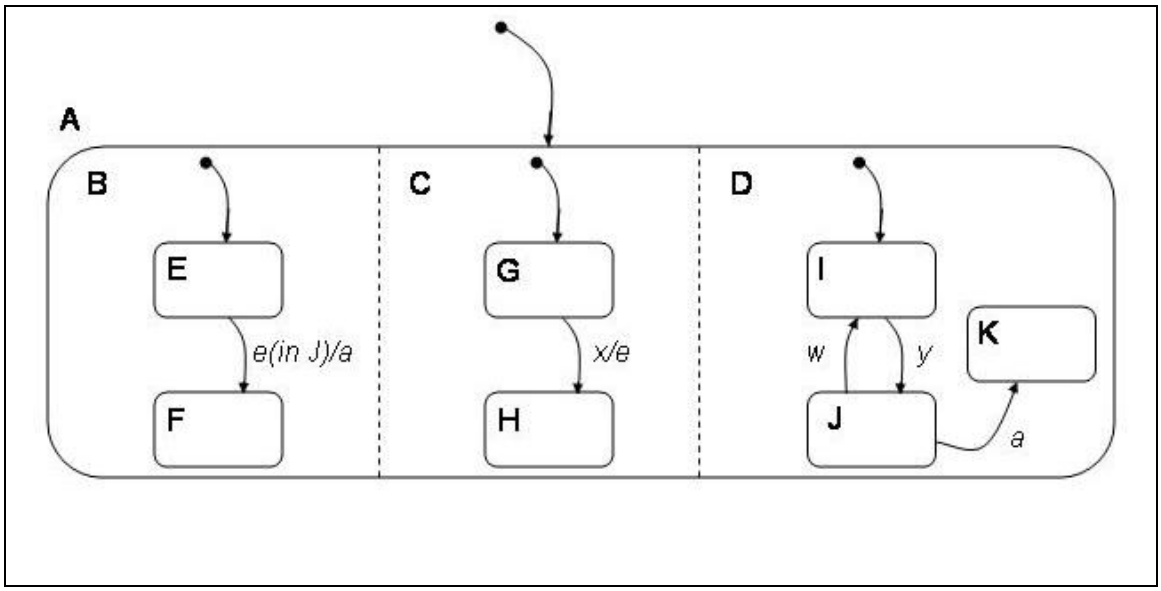


Figure III-6. Examples of Actions and Conditions

In Figure III-6, the system is in a default state (E,G,I). When event *y* occurs, the condition in the I to J transition becomes true, and the system is in state (E,G,J). If event *w* does not occur and event *x* occurs, action *e* will be carried out, which is an event within superstate B. Therefore, the transition between E and F will take place, generating the action *a.* This action is an event within the superstate D and will cause the system transition to state (F,H,K).

An action can also be attached to a state while entering or exiting from that state. The keywords *entry* and *exit* are used for this purpose.

Activities are like actions. However, unlike actions, activities are not instantaneous. They are durable. An activity may be carried out continuously throughout the system's being in the state [2]. Figure III-7 illustrates the concepts.

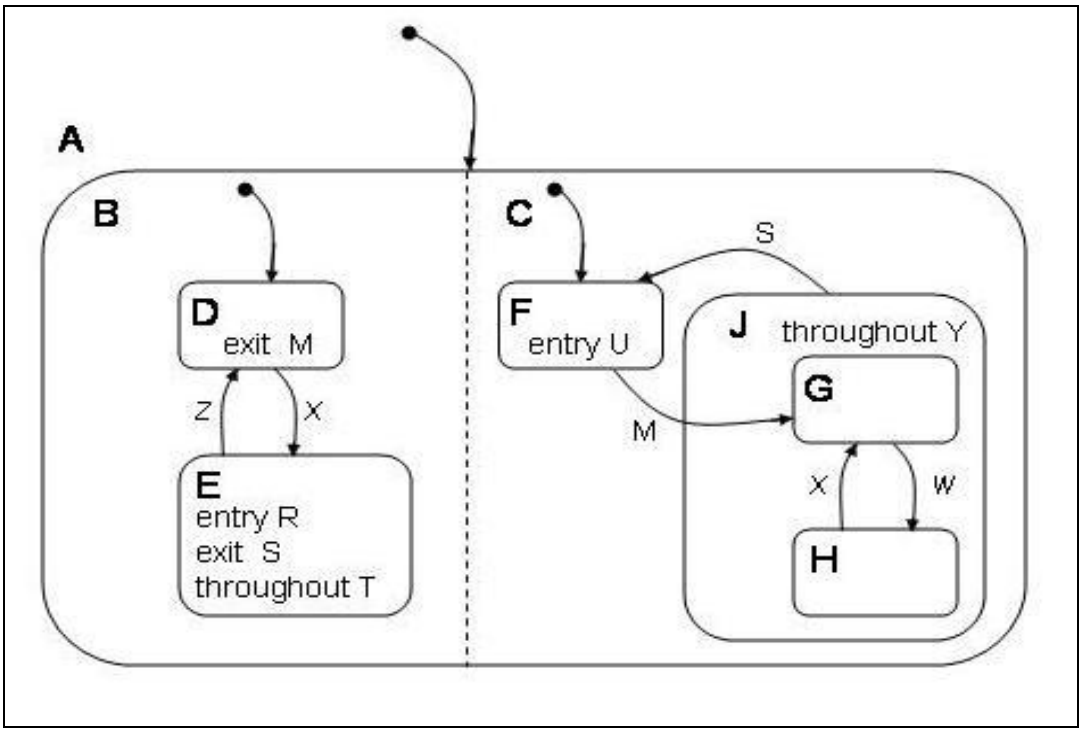


Figure III-7. Examples of Activities

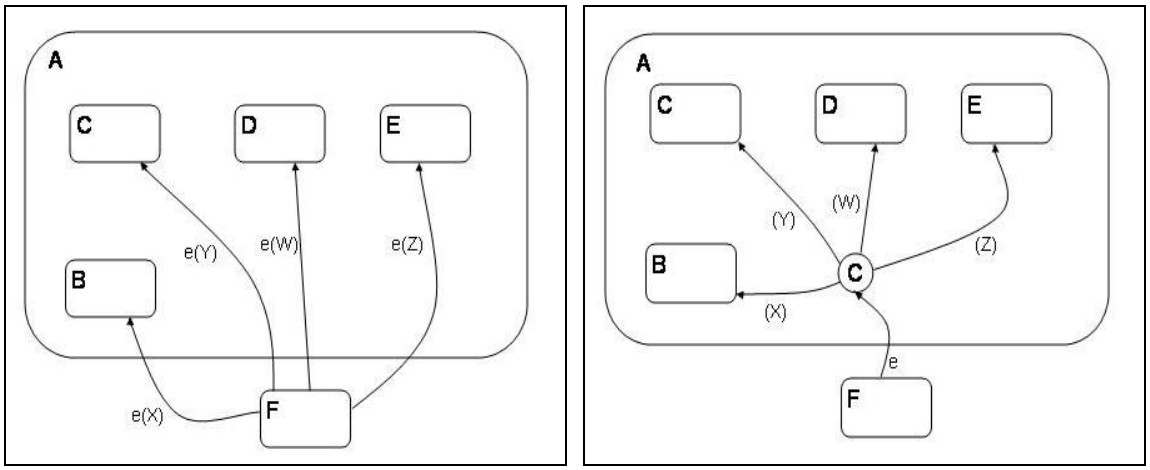
In Figure III-7, T and Y are activities. They carried out by the system when the system is in state (E,J). Action U, which is attached to state F, is accomplished when the system is entering state (D,F). When event *x* occurs, the system actually carries out two actions. The first is due to exiting from state D and the second is because of entering state E. Action S, which is an event between state J and F, is attached to state E on exit.

5. Condition and Selection Entrances

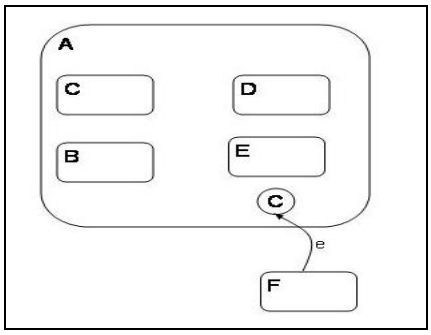
Two other types of circled connectives, other than H (History), are used for abbreviating more complicated entrances to substates of a superstate [2]. The first one is *conditional,* denoted by C, and the second one is *selection,* denoted by S.

Figure III-8 (a) shows a statechart specification without the use of a conditional. Figure III-8 (b) illustrates the specification when the conditional is used. Notice the simplified version in Figure III-8 (c) when the superstate is abstracted.

Selection occurs when a simple event transition to different states based on the value of this simple event. Figure III-9 presents the use of selection in statecharts taken from [2].



(a) (b)



(c)

Figure III-8. Example of a Conditional

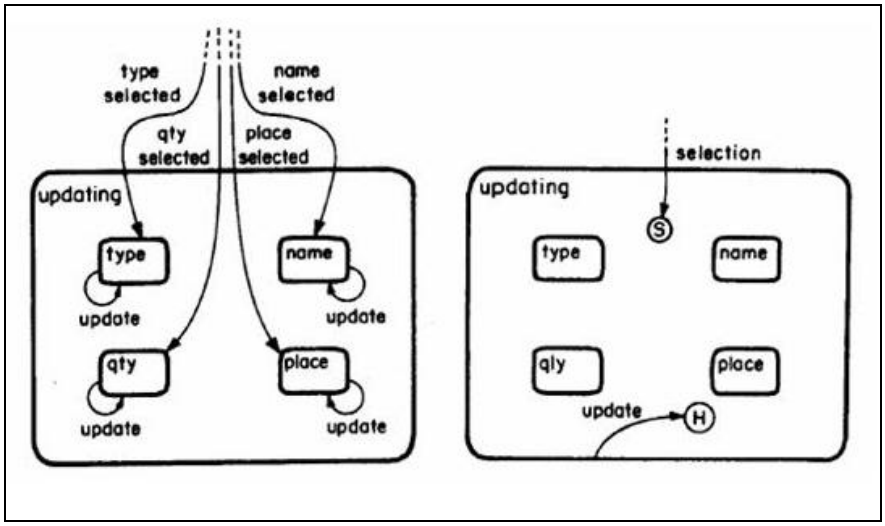


Figure III-9. Example of a Selection [From 2]

6. Delays and Timeouts

In Harel statecharts, time restrictions can be applied to states to some extent. The notation used for time restriction is *timeout(event,number).* This notation represents the event that occurs precisely when the specified number of time units have elapsed from the occurrence of the specified event [2]. A state can also be associated with a lower bound time restriction. The syntax of the specification is $\Delta t_1 < \Delta t_2$, which shows the upper and lower bound at the same time. Figure III-10 illustrates such concepts.

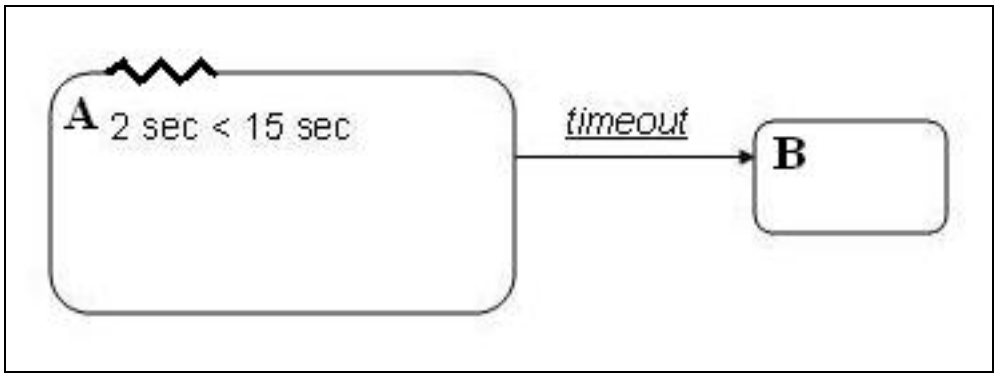


Figure III-10. Examples of a Delay and a Timeout

In Figure III-10, the system has to stay in state A for at least two seconds. State A also has a timeout, which is 15 seconds.

7. Drawbacks of Harel Statecharts

Statechart is one of the most successful visual specification language attempts. However, the language still suffers from some drawbacks:

- Semantics: Statechart is not an official language. Mostly, the semantics are enforced by the tools. The enforced semantics vary among tool vendors.
- Notion of Time: In statecharts, transitions and actions occur instantaneously, which is not realistic for the specification of reactive and real-time systems.
- Race Conditions: If the system is not designed and specified with caution, race conditions may occur. The language is incapable of eliminating such conditions. Consider the example in Figure III-11. Event *x* causes the system transition to state (F,H). These transitions are attached with two actions modifying the value of variable a. However, the assignment is contradictory and the specification is unclear about the final value of a.

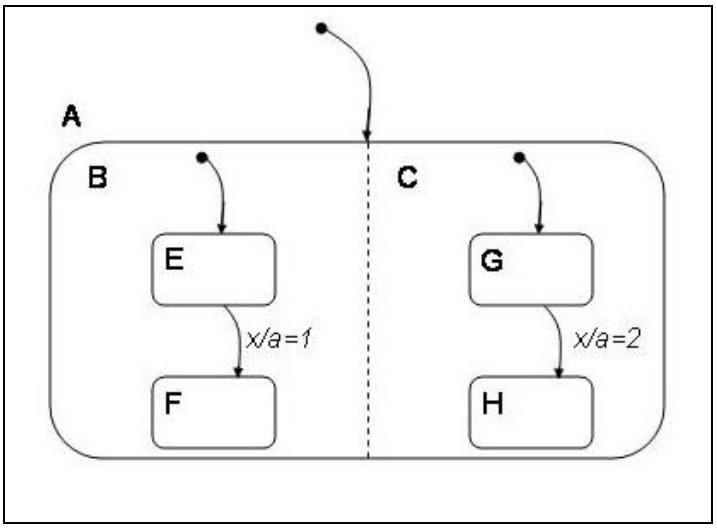


Figure III-11. Example of a Race Condition

B. TEMPORAL LOGIC

Reactive systems are quite different from transformational systems. They are typically, non-terminating, continuously reading input, continuously producing output, and regularly interacting with the environment [24]. Most reactive systems are also real-time systems. For real-time systems, satisfying temporal constraints are crucial. Therefore, the notion of time is important for these systems. Temporal logic has been proposed to specification and verification of program and system behavior, especially for these types of systems. The first significant proposal was made by Pnueli [25, 26].

As discussed earlier, most weapons systems are safety-critical real-time reactive systems and this portion of the thesis will provide a brief introduction to temporal logics.

The specification of a system in temporal logics is generally divided into two aspects: Safety conditions and liveness conditions. Such conditions are described as follows:

- Safety Conditions: These conditions are the type of conditions which the system shall not do. For example, a missile system shall not fire until the operator pushes the unlock safety button.
- Liveness Conditions: These conditions are those which the system should do. For example, the missile shall fire within three seconds after the fire button is pressed.

These conditions help the weapon systems to be better specified. Thus, modeling systems with temporal logics and investigating the correctness of system specification is an essential part of weapon system software development.

1. Classical Logic

The order of a logic theory defines the domain of all formulas described by the logic: Propositional order, first order and higher order.

Propositional logic consists of a set of elementary facts and logic operators. Formulas are built using these facts and operators. Logic operators in classical logic are:

$\neg$: Not

$\wedge$: And

$\vee$: Or

$\Rightarrow$: Conditional

$\Leftrightarrow$: Biconditional

Formulas can be defined in terms of truth tables or by induction on the structure of itself. Formulas take a logical value true ($\top$), or false ($\perp$).

First order logic extends the propositional logic with the introduction of a domain, n-ary relationships on the domain, n-ary predicates associated with the relationships, the universal quantifier $\forall$ (for all), and the existential quantifier $\exists$ (exists). In first order logic, quantified variables must be the elements of the domain and not full predicates. Therefore, quantifiers cannot be varied over predicates. Higher order logic relaxes this constraint and extends the domain modeled by first order logic by allowing adaptation of predicates as quantification variables. [27]

2. Modal Logic

Propositions in classical logic are static. Their value is fixed and time-independent. Therefore, classical logic can only express atemporal (non-time-dependent) formulas. Consider the proposition, “Bob is thirsty”. In classical logic, the proposition will be evaluated true or false for all times. Nonetheless, in real life, the evaluation of this proposition changes over time. Bob might have been thirsty yesterday after jogging. Bob

just drank a soda and might not be thirsty for the next three hours. The example shows that the truth value of the proposition depends on time.

In classical logic, there is only one world. However, in modal logic, there are worlds and the concepts of truth and falsity are not static and immutable. The formulas may be evaluated differently in different worlds.

A modal logic system is defined by <W,R,V> where: W is the set of worlds, $R \subseteq W \times W$ is the reachability relationships between worlds; and V is the evaluation of function for formulas:

$$V : F \times W \rightarrow \{ \top , \bot \},$$

where F is the set of formulas of the modal theory. [27] The modal logic introduces two other operators in addition to classical logic operators: ***L*** (necessary) and ***M*** (possibly). The evaluation function V is defined over the modal operators ***L*** and ***M***, as follows [27]:

$$V(\boldsymbol{M}f,w) = \top \text{ iff } \exists v \in W.wRv \Rightarrow V(f,v);$$

$$V(\boldsymbol{L}f,w) = \top \text{ iff } \forall v \in W.wRv \Rightarrow V(f,v).$$

M and ***L*** can be thought as the existential and universal quantifier defined over the set of reachable worlds from the current world.

Using the modal logic, it is possible to define a relation R, *next instant* over the worlds. This relation makes it possible to define worlds, which are the set of system configurations. The evaluations of a formula within the system model can change in successive time instants. Therefore, modal logic can be used to define temporal behavior.

3. Temporal Logic

Temporal logic is a branch of modal logic. Temporal logic especially deals with the notion of time and order. The definition of temporal logic by NIST [28] is as follows:

> A logic with a notion of time included. The formulas can express facts about past, present, and future states.

Temporal logics extend the classical logic by adding a set of new operators. In temporal logics, the value of a formula is dynamic. Therefore, the evaluation the formula depends on the interpretation and the time. [27]

Generally, temporal logics add four new operators to the classical logic [29].

G, always in the future,

F, eventually in the future,

H, always in the past,

P, eventually in the past.

The operators ***G,F,H,*** and ***P*** can be formally defined as[27]:

$V(\boldsymbol{G}f,t) = \top$ iff $\forall s \in T.t<s \Rightarrow V(f,s)$;

$V(\boldsymbol{H}f,t) = \top$ iff $\forall s \in T.s<t \Rightarrow V(f,s)$;

$\boldsymbol{F}f \equiv \neg\boldsymbol{G}\neg f$;

$\boldsymbol{P}f \equiv \neg\boldsymbol{H}\neg f$.

These operators can express the notion of necessity and possibility; therefore temporal logics are a part of modal logic. There are also other operators that can be added to logics. If the relation < is transitive and nonreflexive, the operators ***until*** and ***since*** can be introduced as follows [27]:

until, with p until q that is true if q will be true in the future and until that instant p will always be true;

since, with p since q that is true if q was true in the past and since that instant p has always been true;

The binary operators ***until*** and ***since*** can be formally defined as:

$V(f_1 \textbf{ until } f_2, t) = \top$ iff $\exists s \in T.t<s \wedge V(f_2,s) \wedge \forall u \in T.t<u<s \Rightarrow V(f_1,u)$;

$V(f_1 \textbf{ since } f_2, t) = \top$ iff $\exists s \in T.s<t \wedge V(f_2,s) \wedge \forall u \in T.s<u<t \Rightarrow V(f_1,u)$.

These operators add more expressiveness to the logics and the previous operators ***G,F,H,*** and ***P*** can be expressed with the operators ***until*** and ***since***:

$\boldsymbol{F}p \equiv \top$ ***until*** p;

$\boldsymbol{P}p \equiv \top$ ***since*** p;

$\boldsymbol{G}p \equiv \neg \boldsymbol{F} \neg p$;

$\boldsymbol{H}p \equiv \neg \boldsymbol{P} \neg p$.

Having distinctive operators for the past and the future eases the specification of the system. In fact, the formulas in temporal logics can easily be shifted to the past or to the future [27].

These operators have other notations as follows:

$\boldsymbol{G} \equiv \Box$,

$\boldsymbol{F} \equiv \Diamond$,

$\boldsymbol{H} \equiv \blacksquare$,

$\boldsymbol{P} \equiv \blacklozenge$,

until $\equiv$ U ,

since $\equiv$ I,

next $\equiv$ o,

prev $\equiv \bullet$.

4. Examples of Temporal Logics

There are many variations on temporal logic: Propositional Temporal Logic (PTL), Choppy Logic, Branching Time Temporal Logic (BTTL), Interval Temporal Logic (ITL), Propositional Modal Logic of Time Intervals (PMLTI), Computational Tree Logic (CTL), Interval Logic (IL), Extended Interval Logic (EIL), Real-Time Interval Logic (RTIL), Logic of Time Intervals (LTI), Real-Time Temporal Logic (RTTL), Timed Propositional Temporal Logic (TPTL), Real-Time Logic (RTL), Tempo Reale ImplicitO (TRIO), Metric Temporal Logic (MTL), and Time Interval Logic with Compositional Operators (TILCO). All these temporal logics can be divided into two categories. The first category consists of the temporal logics without a metric for time. They are PTL, Choppy Logic, BTTL, ITL, PMLTI, IL, CTL, and LTL. The second category are those with a metric for time. EIL, RTIL, RTTL, TPTL, RTL, TRIO, MTL, and TILCO belong

to the second category. The expressiveness of the logics in the first category is not always sufficient for real time system specification, because quantitative temporal constraints cannot be expressed [27]. Other than this categorization, not all of the variants are well-recognized. The most widely-known temporal logics are PTL, BTTL, CTL, and MTL.

Propositional temporal logic (PTL) [25, 26, 30] extends the propositional logic by adding the following temporal operators: always in the future (□), eventually in the future (◊), next (o), and until (U). PTL approaches the system specification from the point of states. The propositions extended with temporal operators specify the system temporal evolution. PTL is a linear event-based logic without providing a metric for time. [27]

Branching time temporal logic (BTTL) [33] is an extension of PTL, which introduces branching in the future. This feature enables BTTL to describe the nondeterministic behavior of systems. Models based on BTTL formulas can be used to build operationally executable state machines. [27] However, the inability to specify quantitative temporal constraints makes the logic unsuitable for real-time system specification.

Computational tree logic (CTL) [31] is a propositional branching time temporal logic unlike many others. The approach of CTL for specifying systems is the notion of several futures, called sequences. The fundamental temporal entity is the point and CTL also does not provide a metric for time.

Metric temporal logic (MTL) [32] extends first order logic with temporal operators: ***G***,***F***,***H***,and ***P*** [27]. The most important aspect of MTL is providing a metric for time. Table III-1 provides a comparison of temporal logics.

Table VII. Comparing Features of Temporal Logics

Logic	Logic order[1]	Fundamental time entity[2]	Temporal structure[3]	Metric for time/ Quantitative temporal constraints[4]	Logic decidability[4]	Deductive system[4]	Logic executability[4]	Ordering events[4]	Implicit, Explicit[5]
PTL	P	P	L	N	Y	Y	Y	Y	I
Choppy	P	P	L	N	Y	(Y)	(Y)	Y	I
BTTL	P	P	B	N	Y	Y	Y	Y	I
ITL	P	I	L	N	(Y)	(Y)	(Y)	Y	I
PMLTI	P	I	L/B	N	(Y)	NA	NA	Y	I
CTL	P	P	B	N	Y	NA	NA	Y	I
IL	P	I	L	N	Y	NA	NA	Y	I
EIL	P	I	L	Y	Y	NA	NA	Y	I
RTIL	P	I	L	Y	Y	NA	NA	Y	(I)
LTI	2^{nd}	I	L	N	Y	Y	NA	Y	(I)
RTTL	1^{st}	P	L	(Y)	N	Y	NA	Y	E
TPTL	P	P	L	Y	Y	Y	NA	Y	(E)
RTL	1^{st}	I	L	Y	N	NA	NA	Y	E
TRIO	1^{st}	P	L	Y	N	Y	(Y)	Y	I
MTL	1^{st}	P	L	Y	(N)	(Y)	NA	Y	I
TILCO	1^{st}	I	L	Y	(Y)	Y	(Y)	Y	I

[1]P= propositional, 1^{st} = first order, 2^{nd} = second order;
[2]P= point, I= interval;
[3]L= linear, B= branching;
[4]N= no, (N)=no in the general case, Y= yes, (Y)=yes in some specific case, NA= not available;
[5]I= implicit, E= explicit.

Table III-1. Comparing Features of Temporal Logics [From 27]

C. TLCHARTS

TLCharts, proposed by Drusinsky [1], is a visual specification language that combines non-deterministic Harel Statecharts, and formal specifications written in Linear-time (metric) temporal logic. TLCharts combines the expressive power of two formalisms and overcomes some of problems of each formalism. Non-deterministic Harel Statecharts are visual and intuitive. Formal specification written in Linear-time (metric) temporal logic has the expressive power to represent complex temporal behavior required by real-time systems. LTL and MTL are textual and the resulting specifications are hard to read and maintain. TLCharts is a true automata-theoretic hybrid of two formalisms with a unified syntax and semantics. Therefore, the hybrid specification language is highly visual and familiar, with special LTL annotation of some transition [1].

Non-deterministic Finite Automate (NFA) can be used as a specification language [34]. The TLChart formalism extends the NFA formalism in two ways [1]:

- NFA formalisms are flat and sequential; however, TLCharts suggests using non-deterministic statecharts.

- TLCharts supports the use of LTL, MTL and TLS [35] assertions along transitions.

TLCharts extend Harel statecharts by [1]:

- Enabling the LTL, MTL or TLS conditioned transitions,
- Supporting non-deterministic behavior,
- Being able to represent good and bad computations with ambiguities resolved based on priorities, and
- Enabling overlapping states.

The relations between TLCharts and its constituents are given in Figure III-12.

Armor-plating a specification is the activity of over-specification with the purpose of providing additional assurance that a specific requirement is satisfied. TLCharts offer such an opportunity to fully specified TLCharts or statecharts by adding temporal conditions. An example of armor-plating a system specification using TLCharts is found in [36].

The motivation behind the TLCharts specification language is the concern for effective, early phase specifications before system design and implementations. In most cases, formal specifications of other types are used to analyze the correctness properties of an existing system. The detailed syntax and semantics of TLCharts can be found in [1].

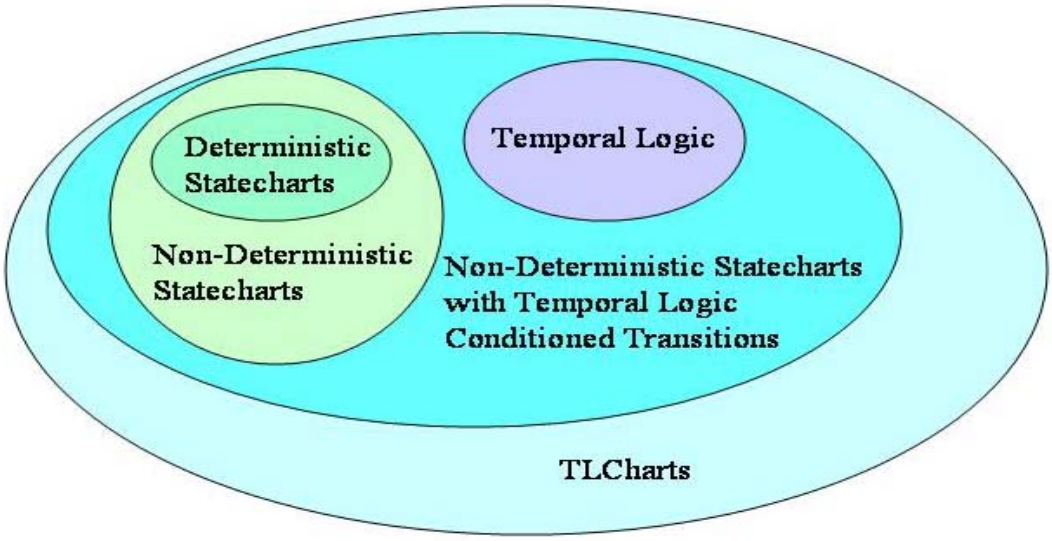


Figure III-12. Relations between TLCharts and its Constituents (Syntax)

1. An Infusion Pump Keypad Control Example

The infusion pump keypad control example is taken from [37]. The example consists of four conditions: **infusionBegin, infusionEnd, keyPressed,** and **alarm**. The associated requirement is as follows:

> R1: Between every **infusionBegin** and an **End-condition** session, a **keyPressed** must be repeatedly sensed within 2-minute intervals. Otherwise, an alarm must sound within 10 seconds and until **keyPressed** is sensed. Once the alarm sounds according to this specification than the assertion has succeeded and no more alarms are permitted. The **End-condition** is defined **infusionEnd** being sensed until **infusionBegin** is sensed.

The following MTL assertion is an attempt to capture the previous requirement R1:

L1: □ (infusionBegin =>

L2: (((infusionBegin ∨ keyPressed) =>

L3: ((□ ¬alarm) ∧

L4: ((O $\Diamond_{\leq 120}$ keyPressed)

L5: ∨ (¬keyPressed $U_{[120,130]}$

L6: (alarm U

L7: (keyPressed∧ □¬alarm)))

L8:)

L9:)

L10:) U (infusionEnd U

L11: (infusionBegin ∧ infusionEnd))

L12:))

In the above MTL assertion, line L1 starts the session. Lines L2 and L4 together require a recurring **keyPressed** event that is to be sensed every two minutes. Line L3 ensures that the alarm is not continuous during those two minutes. Lines L5, L6 and L7

ensure an alarm within 10 seconds at the end of the two minutes period and until **keyPressed** is sensed. There will be no alarm afterwards. Line L10 and L11 are for the **End-condition** defined as **infusionEnd** being sensed until **infusionBegin** is sensed.

As discussed in [37], the MTL assertion suffers from several deficiencies. First of all, even though the natural language requirement is easier to understand, the previous attempt to formalize the requirement using MTL is arguably non-trivial. Some terms used in the MTL assertion are confusing, such as "**infusionEnd U (infusionBegin** $\wedge$ **infusionEnd)**". Also, the MTL specification does not forbid an alarm while not in session. Such constraint is included in Figures III-13 and III-14. Figure III-13 is a Harel statechart specification of the assertion while Figure III-14 is the corresponding TLChart specification.

The MTL specification might fail under certain scenarios, which are discussed in detail in [37]. In short, it can be argued that temporal logic requirements are hard to read, write and maintain. TLChart specification attempts to overcome such difficulties by bringing a visual component to the formalism.

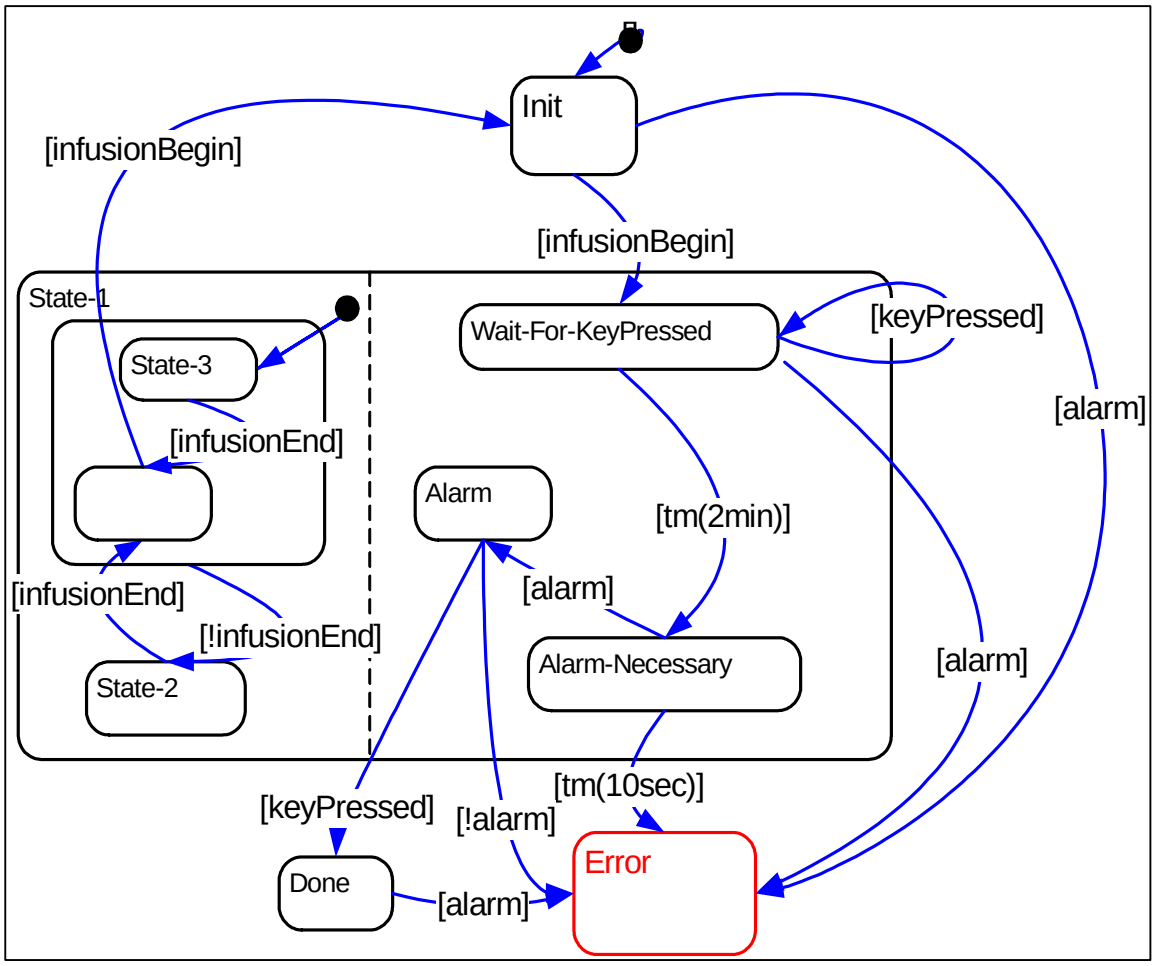


Figure III-13. Deterministic Harel Statechart Specification for Requirement R1 [From 37]

In the Harel statechart specification, the transition from state *Init* to *Error* ensures that there should be no *alarm* during sessions. Such *Error* state is also included in the TLChart specification for the requirement R1 in Figure III-14.

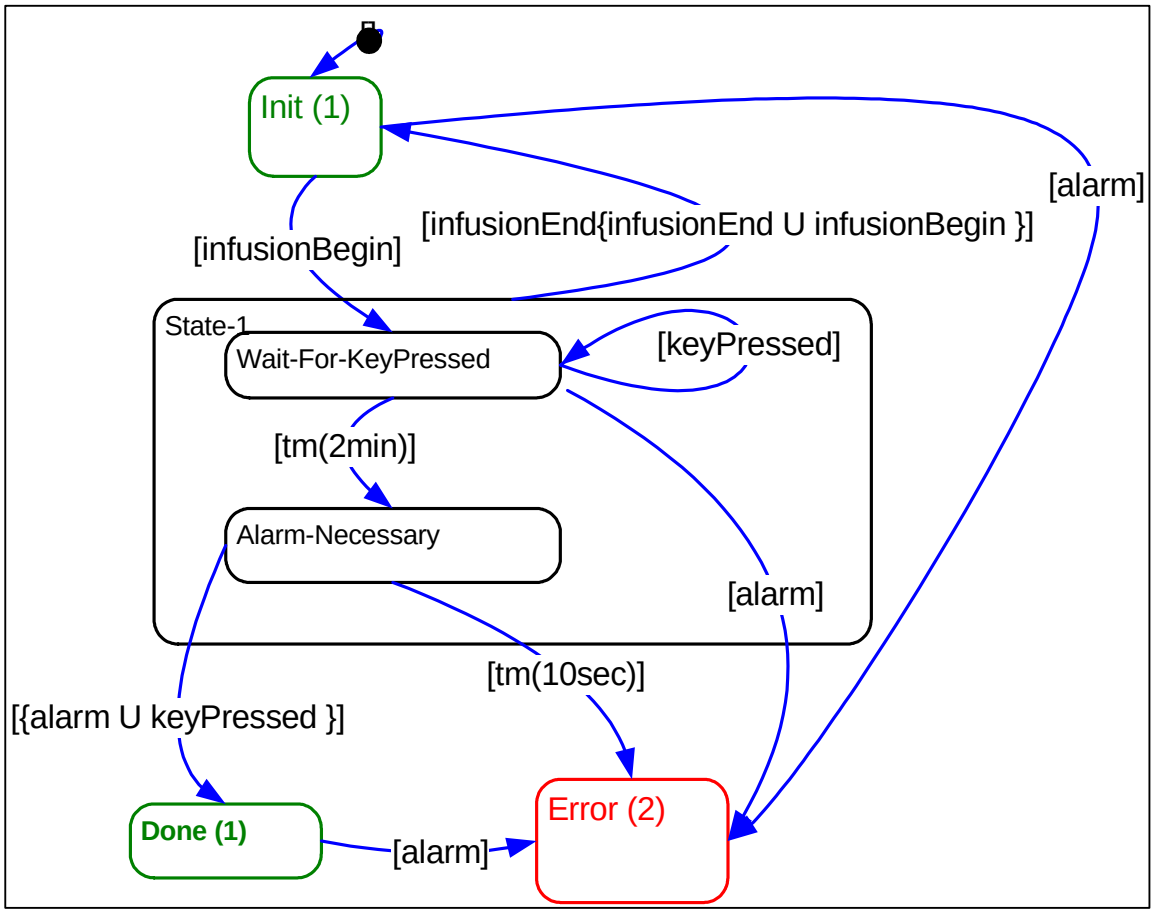


Figure III-14. TLChart Specification for Requirement R1 [From 37]

Both Figures III-13 and III-14 are in fact legal TLCharts. Since Harel statecharts are a special case of TLCharts, so are LTL and MTL. The specifications in both figures solve the problems previously mentioned. The TLChart differs in two main ways from a deterministic statechart [37]:

1. Some transitions include LTL, MTL and TLS conditions. For example, the transition labeled "**alarmU keyPressed**".
2. TLChart specification is non-deterministic, which may exist in two ways:
   a. Two or more propositional conditions may become true simultaneously. For example in Figure III-14, **alarm** and **keyPressed** may become true at the same time, creating ambiguity. While this form of non-determinism is undesirable, it is easily avoidable using logic prioritization or by using events, guaranteed to be mutually exclusive.
   b. Two or more transitions with temporal conditions may be traversed at the same time.

Harel statecharts must be deterministic when used for a specification. Otherwise, the specification is ambiguous. Achieving a correct deterministic behavior is the most critical part of the implementation process.

IV. A CASE STUDY: KTORP

A. KTORP INTRODUCTION

KTorp is an artificially created submarine-launched homing torpedo. Although it is an artificial torpedo, the specification of KTorp captures some important dynamic characteristics of real homing torpedoes. Since this is an unclassified study, a real torpedo example cannot be used. Therefore, detailed classified specifications of similar real torpedo systems are not mentioned in this study. The rest of the specifications are deviated on purpose from real examples so as not to reveal classified information. In this sense, KTorp is abstracted and simplified from real torpedo examples. Even with its current simplified specification, KTorp adequately serves as a weapon system example for the purpose of this study.

KTorp is a submarine-launched homing torpedo for underwater targets. Many torpedoes have an umbilical cord attached to its mother ship. This cord (in other words wire) enables the torpedo to be guided by the submarine personnel after the torpedo launched. However, the cord limits the maneuverability of the submarine. To simplify the case study, KTorp is specified without an umbilical cord. In this sense, KTorp is a "launch-and-forget" type of homing torpedo. KTorp can operate in any sea condition and has five main sections:

- **Homing Head Section:** This section is found in the head of KTorp. It contains a sonar device, transceivers and the necessary wiring.
- **Warhead Section:** The warhead with an exploder is the main parts of this section. The ignition of explosives in the warhead is triggered by a proximity switch. Therefore, KTorp must hit its target in order to explode and destroy.
- **Battery Section:** KTorp is a battery-powered torpedo just like most real torpedoes. The battery section provides power for the navigation and the electronics. KTorp will try to search and destroy its target until the battery depletes.
- **Control Section:** All the necessary electronic control logic is found in this section. The torpedo has a central embedded computer and various embedded controllers to guide itself to the target and control all the support functions.

- **Motor Section:** This section contains the engine and the propellers. The engine is also controlled by the central embedded computer found in the control section. The engine is powered by the battery. The propellers are located at the back of the torpedo.

Figure IV-1 shows an illustration of KTorp sections.

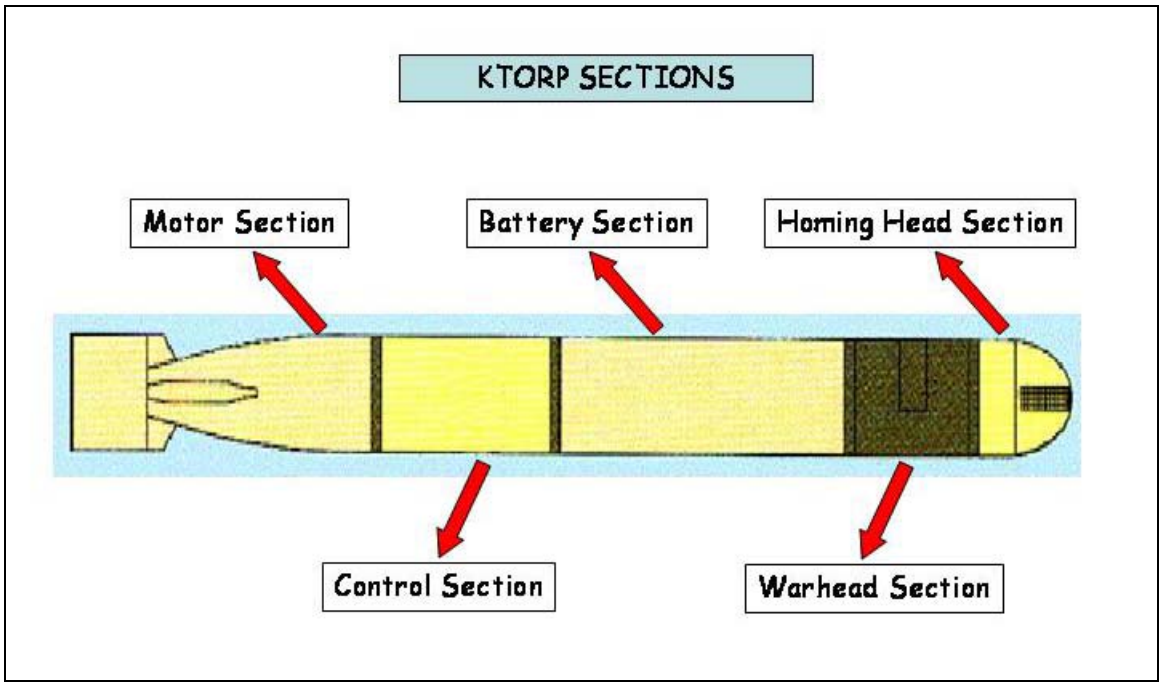


Figure IV-1. Illustration of KTorp Sections

B. KTORP TECHNICAL SPECIFICATIONS

Ktorp technical specifications are as follows:

TS.1. The torpedo has two different speeds:

- High Speed: 42 Knots
- Low Speed: 30 Knots

TS.2. The navigation depth of KTorp is between 35 ft. and 2,000 ft.

TS.3. The approximate torpedo run time, which depends on the state of the battery, is about 30 minutes at low speed and 15 minutes at high speed.

TS.4. KTorp has two different search modes:

- Snake Search Mode
- Circular Search Mode

The details of these modes will be given in the dynamic specification of KTorp.

TS.5. KTorp can be set to enable at a particular distance from the mother ship. This distance is called “Enable Distance”. The enable distance can be configured between 500 yards and 3,000 yards before launch.

C. SAFETY-RELATED SPECIFICATIONS

Most weapon systems are safety critical systems. Ktorp is one such system., Therefore, this section specifies the requirements that are safety related.

SR.1. If KTorp is unable to find its target, it should not explode in any case. At the end of run, the power to the warhead is cut and the torpedo will be disarmed.

SR.2. KTorp should not explode in any case before it reaches the enable distance.

SR.3. KTorp should not navigate above 35 feet in any case.

SR.4. If KTorp reaches a depth below 2,000 feet, it should not explode in any case. Therefore, the torpedo must be disarmed after reaching this depth. (The sea pressure below 2,000 feet may damage the torpedo hull and decrease the probability of it functioning correctly.)

SR.5. KTorp is disarmed after 35 minutes of run time in any case.

D. DYNAMIC BEHAVIOR OF KTORP

Like every other weapon system, the goal of KTorp is destroying the target. In order to reach this goal, KTorp searches, attacks and destroys its target. These behaviors are differentiated using torpedo run phases. The dynamic behavior of KTorp consists of three basic phases.

1. Enable Phase

The goal of this phase is to ensure the safety of the mother ship. In the enable phase, KTorp is disarmed and its sonar is inactive. Thus, the torpedo is unable to detect signals and attack its mother ship by any means. During this phase, the torpedo will follow a straight path until it reaches the enable distance. The enable phase ends with an internal signal generated by a unit called “Inertial Measurement Unit” (IMU). This unit is responsible for calculating the distance traveled.

2. Search Phase

The goal in this phase is to detect a potential target. After the enable phase, the torpedo enters the search phase. The sonar and transceivers become active and the torpedo begins to search for its target. This search has a predefined specification determined by KTorp search mode.

3. Attack Phase

The goal in this phase is to focus on a potential target. KTorp narrows its search space and increases the sensitivity of signal reception. This phase is the final phase before the final attack. If KTorp is unable to find the target within one minute, then it will turn back to the search phase.

E. SEARCH MODES

As mentioned in the technical specifications, there are two different search modes. The reason for two different search modes lies in the solution of a fire control problem. When a submarine detects a potential target, the fire control party (the submarine personnel responsible for the weapon systems) enters the estimated parameters of the target into the fire control system. Estimated parameters define the movement of the potential target. Then, the fire control system analyzes the parameters and sets up a fire control problem. The sensors monitor the potential target and feed the fire control system with the updated parameters. After the fire control system reaches a solution for the fire control problem, the submarine commander decides on the search mode of the torpedo. Then, the torpedo is a launched.

The first type of search mode is called “Snake Search” because of the similarity between a snake’s movement and the navigation pattern of KTorp in this search mode. This type of search mode is used when the depth of the target is known but the distance is unknown.

The second type of search mode is called “Circular Search”. Again, the name of the mode reveals the navigation pattern of KTorp. In this mode, the first 1,000 yards of the navigation resembles the snake search except for the specification of the signal reception. After this navigation, KTorp begins to dive and search for its target by circling with a diameter of 1,000 yards. When the torpedo reaches the depth limit (2,000 ft), it

begins to rise with the same pattern. The circular search mode is used when the distance of the potential target is known but the depth is not known.

The exact patterns of search modes are given below in steps.

1. Snake Search

SS.1. KTorp will dive or rise to its search depth with a straight navigation path. KTorp is in enable phase until it reaches enable distance.

SS.2. When KTorp reaches the enable distance, it will arm itself and its sonar will become active. At this point, KTorp enters the search phase.

SS.3. In the search phase, KTorp's navigation is just like a snake. First, KTorp will steer 15° right angle ("starboard" in Navy lingo) and navigates 300 yards. Then, KTorp will steer 30° left angle ("port" in Navy lingo) and 30° right angle after navigating 300 yards in each turn. This pattern continues until KTorp enters the attack phase.

SS.4. When the torpedo receives two signals with a five msec. pulse length within five seconds, KTorp enters the attack phase. (The noise in the environment is filtered by the sonar and these signals are certain to be generated by a potential target with a high probability)

SS.5. In the attack phase, KTorp focuses on the potential signal source. In this mode, first the torpedo steers 7.5° right angle and navigates 150 yards. Then, KTorp will steer 15° left and 15° right angle after navigating 150 yards in each turn.

SS.6. When the torpedo receives three signals with 2.5 msec. pulse length within three seconds, KTorp decides that the signal source is the real target and attacks the target.

SS.7. If KTorp does not receive the signals specified in step SS.6 within one minute, it returns to the search phase.

S.S.8. While KTorp is attacking the target, it follows a straight path until it destroys the target or the torpedo run ends.

The illustration of a snake search pattern is given in Figures IV-2 and IV-3.

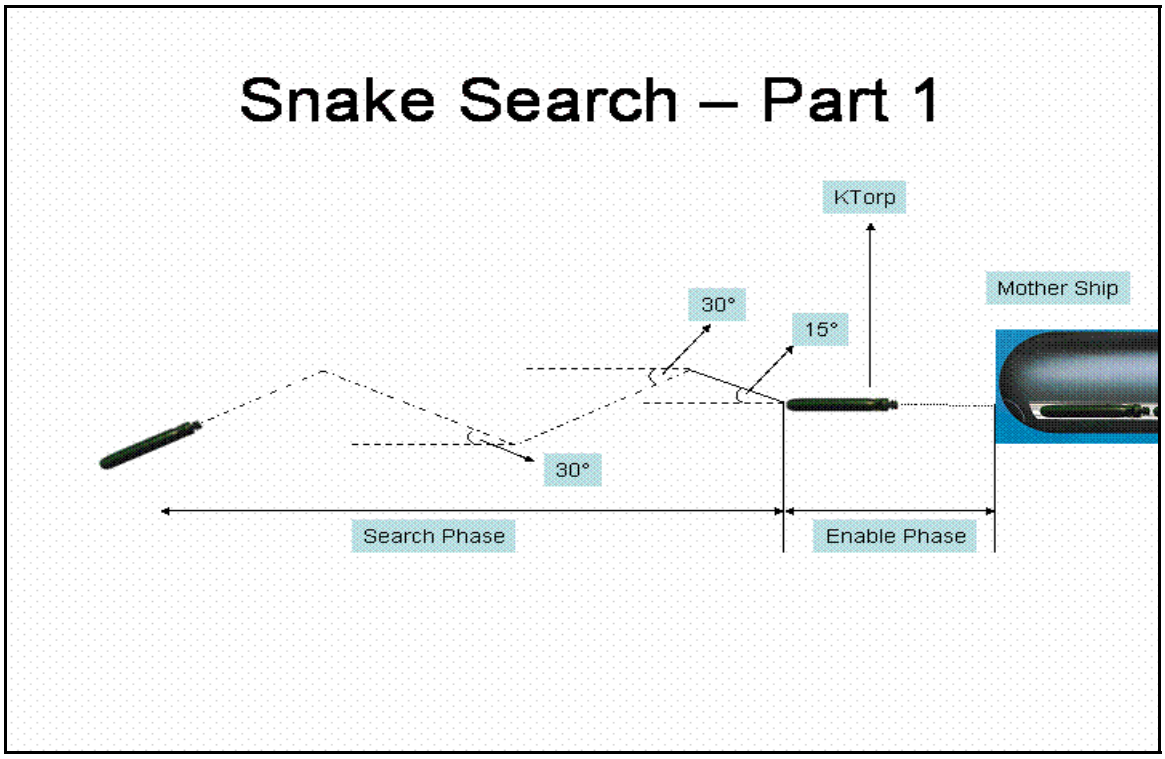


Figure IV-2. Snake Search Pattern Part 1

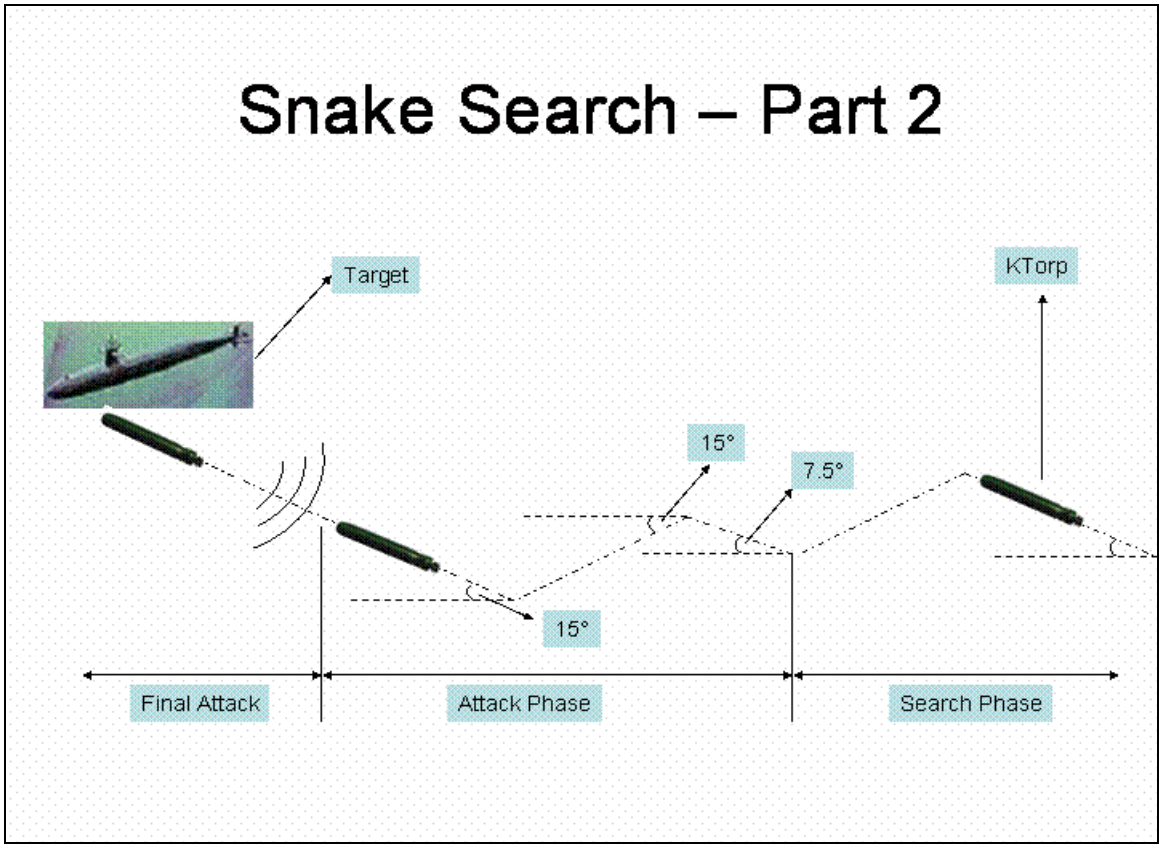


Figure IV-3. Snake Search Pattern Part 2

2. Circular Search

CS.1. KTorp will dive or rise to its search depth with a straight navigation path. KTorp is in the enable phase until it reaches enable distance.

CS.2. When KTorp reaches the enable distance, it will arm itself and its sonar will become active. At this point, KTorp enters the search mode.

CS.3. In the search phase, the navigation pattern of KTorp is exactly the same with snake search for the first 1000 yards. At this point, KTorp begins to circle by steering left. The approximate diameter of the search circle is 1,000 yards. KTorp steers 2° left angle at every second. While circling, the torpedo dives with 7.5° down angle. After reaching the 2,000 ft depth limit, the torpedo rises to the search depth with 7.5° upwards angle and continues to circle.

CS.4. In the search phase, when KTorp receives two signals with 2.5 msec. pulse length within four seconds, it enters the attack phase.

CS.5. The navigation pattern in the attack phase is the same with the search phase as specified in step CS-3.

CS.6. When KTorp receives four signals with 2.5 msec. pulse length within two seconds, it decides that the signal source is a real target and attacks towards the location of the signal source.

CS.7. If KTorp does not receive the signals specified in step 6, it returns to search phase.

CS.8. While KTorp is attacking the target, it follows a straight path until it destroys the target or the torpedo run ends.

The illustration of circular search pattern is given in Figures IV-4 and IV-5.

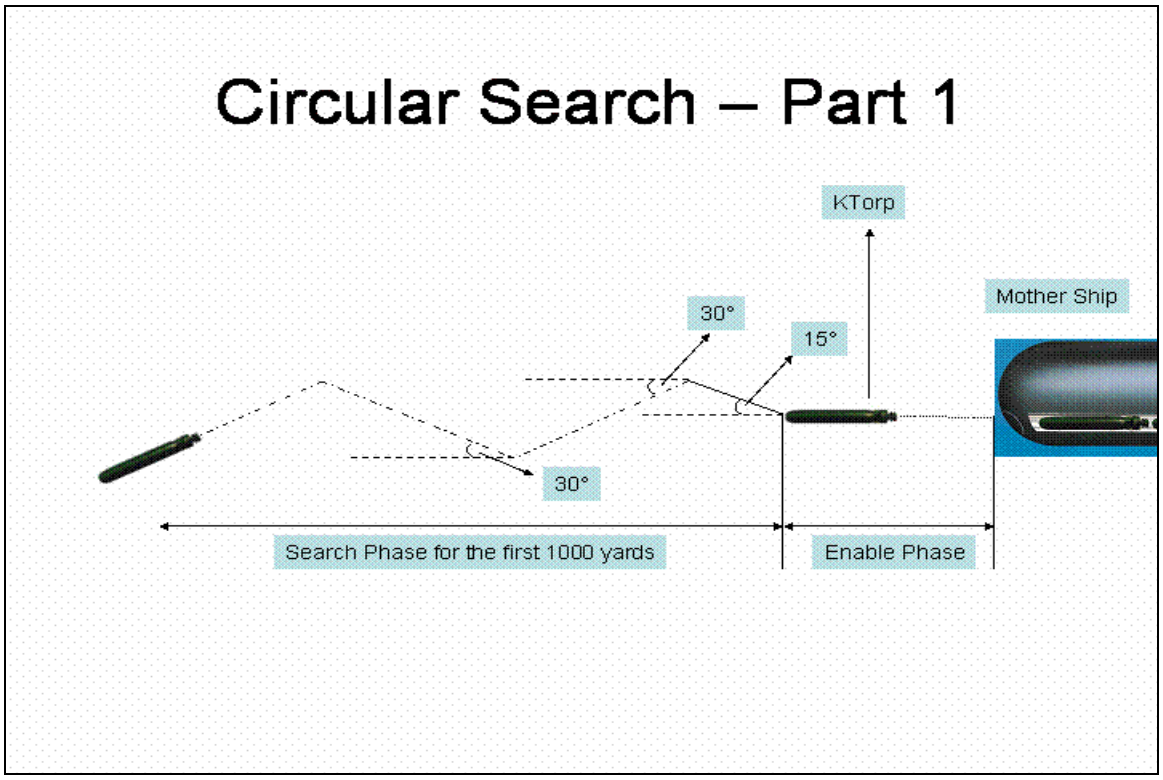


Figure IV-4. Circular Search Pattern Part 1

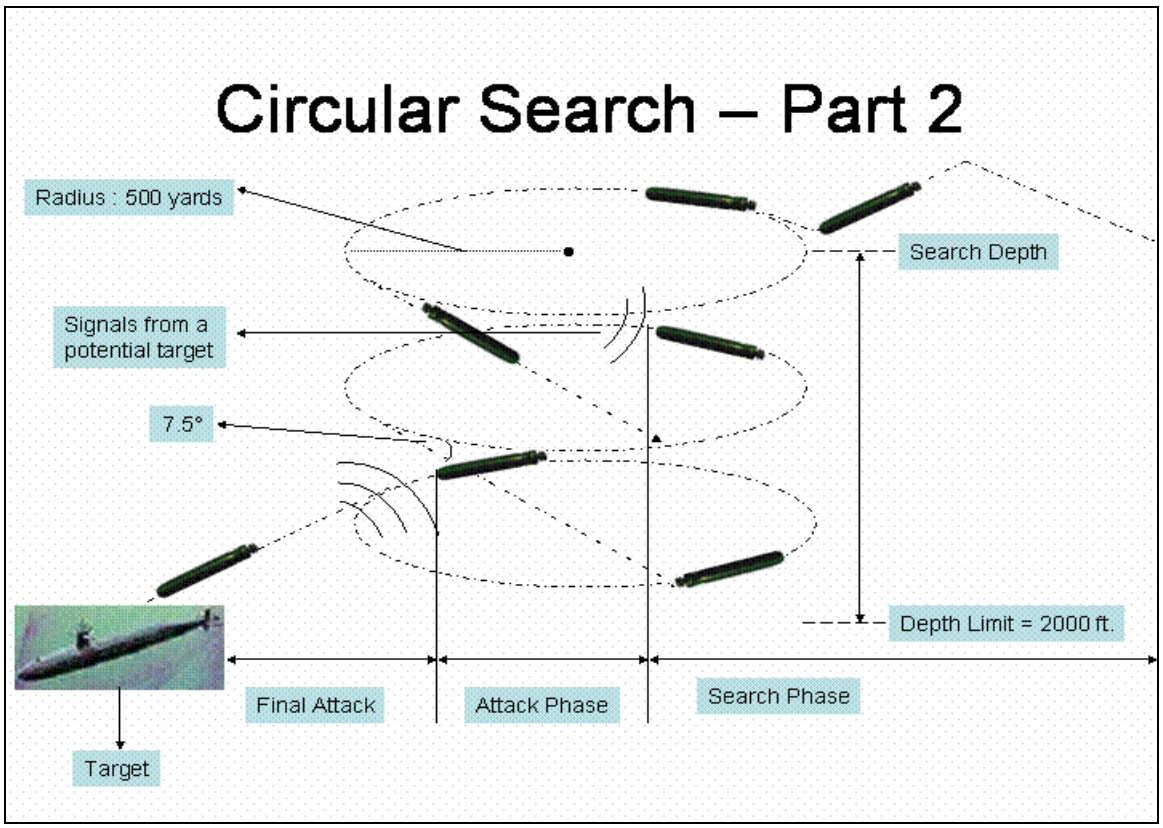


Figure IV-5. Circular Search Pattern Part 2

F. HIGH LEVEL USE CASES

Developing use cases is one of the most important activities defined by the Unified Process. Basically, they provide a starting point for analyzing and developing systems. A use case is a generalized scenario of a system's particular functionality. A high level use case explains the system's functionality by abstracting many details. A high level use case may be refined to provide more detail and understanding of the system's behavior. Therefore, more refined use cases may be developed at different abstraction levels whenever necessary. In this case study, the focus is in high level. Thus, only high level use cases will be developed.

As mentioned in the KTorp introduction, KTorp is a simplified version of similar real torpedo systems. Many features and specifications are abstracted. This study only focuses on the dynamic behavior of KTorp, which is searching and destroying the target. In fact, the logic design of the dynamic behavior is the one of the most challenging parts of torpedo designs. The correct implementation of this behavior is crucial for the torpedo system development. Therefore, only use cases relevant to dynamic behavior is included.

At the highest level, two distinctive use cases can be easily identified. They are basically the two search modes of KTorp: Snake Search and Circular Search. The use cases also include the alternative steps because these use cases will provide the inputs to specify the system using TLCharts. Although it is simple, the use case diagram is provided in Figure IV-6 to follow the development process.

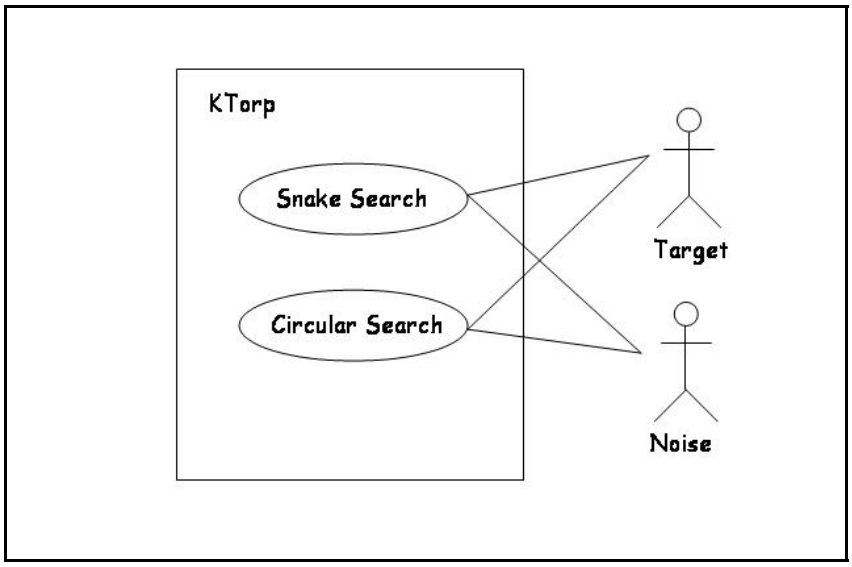


Figure IV-6. Use Case Diagram of KTorp

1. Snake Search Use Case

Version: 1

Modifications and Rationale from Previous Versions: None.

Use Case Identifier and Name: UC-1, Snake Search

Primary Actors: Target

Priority: High-level primary and essential

Preconditions:

1. The pre-launch torpedo system checks are normal.
2. The parameters for enable distance, search depth and snake search mode are successfully given to the torpedo by the fire control system.
3. The torpedo is configured for the snake search mode.
4. The torpedo is launched successfully from the mother ship.

Postconditions:

1. The torpedo either destroys the target or the battery dies and the torpedo sinks silently. Unless KTorp hits the target, no explosion should occur.

Main Scenario:

1. After the launch, KTorp is in the enable phase. The torpedo dives or rises to the search depth. It is disarmed. The sonar and related devices are inactive.
2. The depth sensor, feeds the torpedo control logic with depth feedback throughout the complete run.
3. KTorp travels its enable distance in a straight line.
4. Inertial Measurement Unit (IMU) feeds the distance traveled to the torpedo control logic.
5. KTorp continuously monitors the feedback from the IMU in order to satisfy safety requirements.
6. The Enabler sends the enable signal to the torpedo control logic when it reaches the predetermined enable distance.
7. The torpedo arms itself. The sonar and related devices become active. After these operations are completed, KTorp enters the search phase.
8. In the search phase, first KTorp steers 15° right. Then, it steers left and right with a 30° angle. After every turn, it navigates 300 yards. This movement continues until KTorp enters the attack phase.
9. The target in the environment generates signals.
10. The target signals are captured by the transceivers of KTorp throughout the torpedo run.
11. The noise in the environment creates signals similar to a target signature throughout the torpedo run.
12. The noise signals are captured by KTorp transceivers.
13. The torpedo analyzes the signals within the environment to determine if the signals are originated from a target or from the noise in the environment.
14. When KTorp receives two signals with five msec. pulse length within five seconds, it enters to the attack phase.
15. The torpedo concentrates its search and modifies its navigation pattern as follows: First KTorp steers 7.5° right. Then, it steers left and right with a 15° angle. After every turn, it navigates 300 yards. This movement continues until KTorp attacks the target.
16. When the torpedo receives three signals with 2.5 msec. pulse length within two seconds, KTorp switches to high speed and attacks its target in a straight line.

Alternative Flow of Actions:

*16. If KTorp does not receive the attack signal pattern (three signals with 2.5 msec. pulse length within two seconds) for one minute, KTorp will return to the step 8 (search phase)

* Whenever the torpedo battery runs down, it will disarm itself.

2. Circular Search Use Case

Version: 1

Modifications and Rationale from Previous Versions: None.

Use Case Identifier and Name: UC-2, Circular Search

Primary Actors: Target

Priority: High-level primary and essential

Preconditions:

1. The pre-launch torpedo system checks are normal.
2. The parameters for enable distance, search depth and snake search mode is successfully given to the torpedo by the fire control system.
3. The torpedo is configured for circular search mode.
4. The torpedo is launched successfully from the mother ship.

Postconditions:

1. The torpedo either destroys the target or the battery dies and torpedo sinks silently. Unless KTorp hits the target, no explosion should occur.

Main Scenario:

1. After launch, KTorp is in the enable phase. The torpedo dives or rises to the search depth. It is disarmed. The sonar and related devices are inactive.
2. The depth sensor, feeds the torpedo control logic with depth feedback throughout the complete run.
3. KTorp travels its enable distance in a straight line.
4. Inertial Measurement Unit (IMU) feeds the distance traveled to the torpedo control logic.
5. KTorp continuously monitors the feedback from the IMU in order to satisfy safety requirements.

6. The Enabler sends the enable signal to the torpedo control logic when it reaches the predetermined enable distance.

7. The torpedo arms itself. The sonar and related devices become active. After these operations are completed, KTorp enters into search phase.

8. In the first part of the search phase, the navigation pattern is the same with snake search mode for 1,000 yards. First, KTorp steers 15° right. Then, it steers left and right with a 30° angle. After every turn, it navigates 300 yards. This movement continues until KTorp navigates 1,000 yards.

9. In the second part of search phase, KTorp begins circling. The diameter of the circles is 1,000 yards. The torpedo steers 2° right angle at every second. While circling, KTorp also dives with a 7.5° down angle. When it reaches the depth limit, which is 2,000 ft., KTorp begins to rise with a 7.5° upwards angle. This rising continues until KTorp reaches the upper depth limit, which is 35 ft. Then, the torpedo begins to dive again and this circling pattern continues until the torpedo goes for its final attack.

10. The target in the environment generates signals.

11. The target signals are captured by the transceivers of KTorp throughout the torpedo run.

12. The noise in the environment creates signals similar to a target signature throughout the torpedo run.

13. The noise signals are captured by KTorp transceivers.

14. The torpedo analyzes the signals within the environment to determine if the signals are originated from a target or from the noise in the environment.

15. When KTorp receives two signals with 2.5 msec. pulse length within four seconds, it enters into the attack phase.

16. In the attack phase, the torpedo concentrates its search but does not modify its navigation pattern as in the snake search mode. When KTorp receives four signals with 2.5 msec. pulse length within two seconds, KTorp switches to high speed and attacks its target in a straight line.

Alternative Flow of Actions:

*16. If KTorp does not receive the attack signal pattern (four signals with 2.5 msec. pulse length within two seconds) for one minute, KTorp will return to the step 9 (search phase)

* Whenever the torpedo battery runs down, it will disarm itself.

G. SYSTEM SEQUENCE DIAGRAMS

Developing system sequence diagrams is another important activity within the Unified Process. A system sequence diagram explains the interactions between objects at the system level. Its focus is in interactions incorporating the notion of time. High level use cases provide the necessary input to develop system sequence diagrams. They can either be used to model the generic interactions or specific instances of an interaction. Modeling generic interactions provides all possible paths whereas modeling specific instances of an interaction provides just one path through the interaction. In sequence diagrams, object instances are arranged horizontally and time runs vertically [41]. Two system sequence diagrams are developed for this case study at the highest level. Each of them explains one use case. Figure IV-7 shows the snake search sequence diagram and Figure IV-8 shows the circular search sequence diagram.

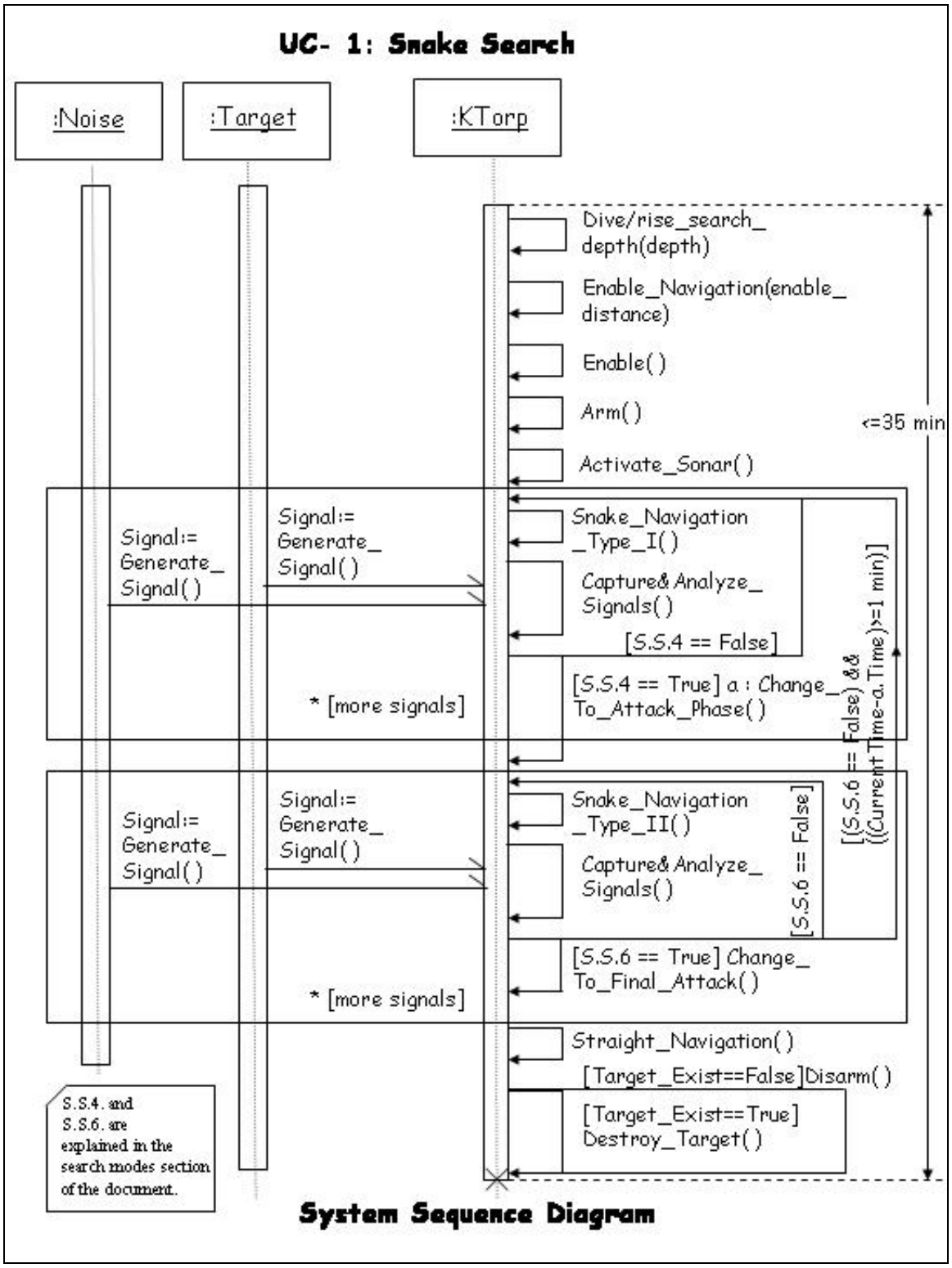


Figure IV-7. System Sequence Diagram for KTorp Snake Search

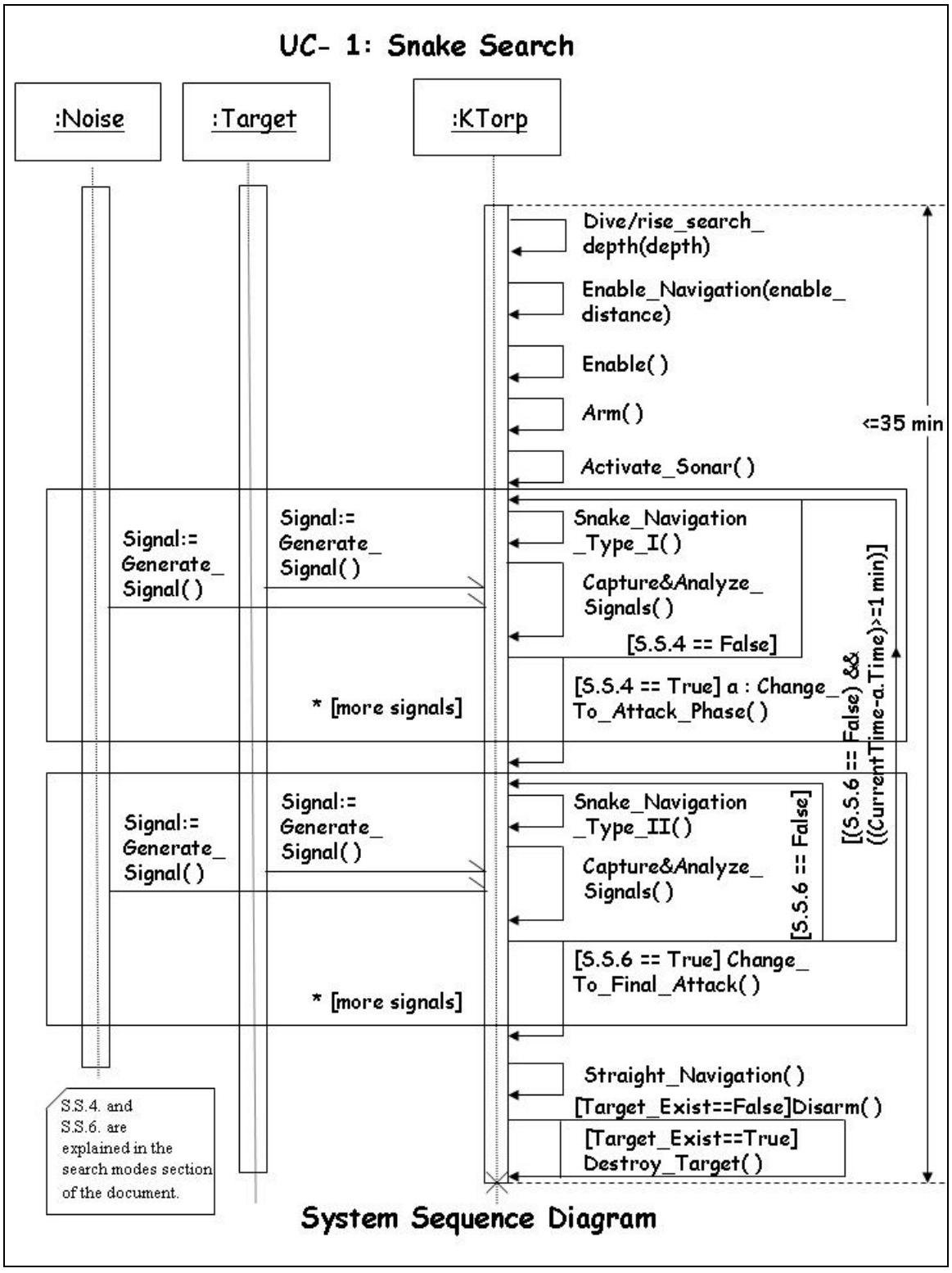


Figure IV-8. System Sequence Diagram for KTorp Circular Search

H. STATECHARTS

Today, almost every software development follows an iterative approach. This practice is also recognized in this study and several iterations are used to develop the KTorp statechart specification.

In the first iteration, the system is analyzed at the highest level. Using natural language specifications provided earlier, the KTorp statechart specification should include four main states:

- Enable_Phase State
- Search_Phase State
- Attack_Phase State
- Final_Attack State

Identifying the main states in a system is crucial for system statechart specification. Therefore at the highest level, the details of the specification are abstracted by substituting the events and the guards with the letters in the alphabet. The first iteration of the specification is shown in Figure IV-9. Note that this specification accounts for both search modes.

KTorp has also safety-related requirements, which were listed earlier. For example, the safety-related natural language specification SR.5 is as follows:

SR.5. KTorp is disarmed after 35 minutes of run time in any case.

At the highest level, some of these safety-related requirements are incorporated in the highest level statechart by adding a transition from every main state labeled "E,F,G,H/Disarm". E, F, G, and H are the necessary events and conditions. Disarm is the action simplified to represent all the tasks related to disarming the torpedo.

In the second iteration, the snake search mode is analyzed. The statechart specification starts with "Enable_Phase" state. This phase is defined in natural language specification SS.1 as follows.

SS.1. KTorp will **dive or rise to its search depth** with a straight navigation path. KTorp is in the enable phase until it **reaches enable distance**.

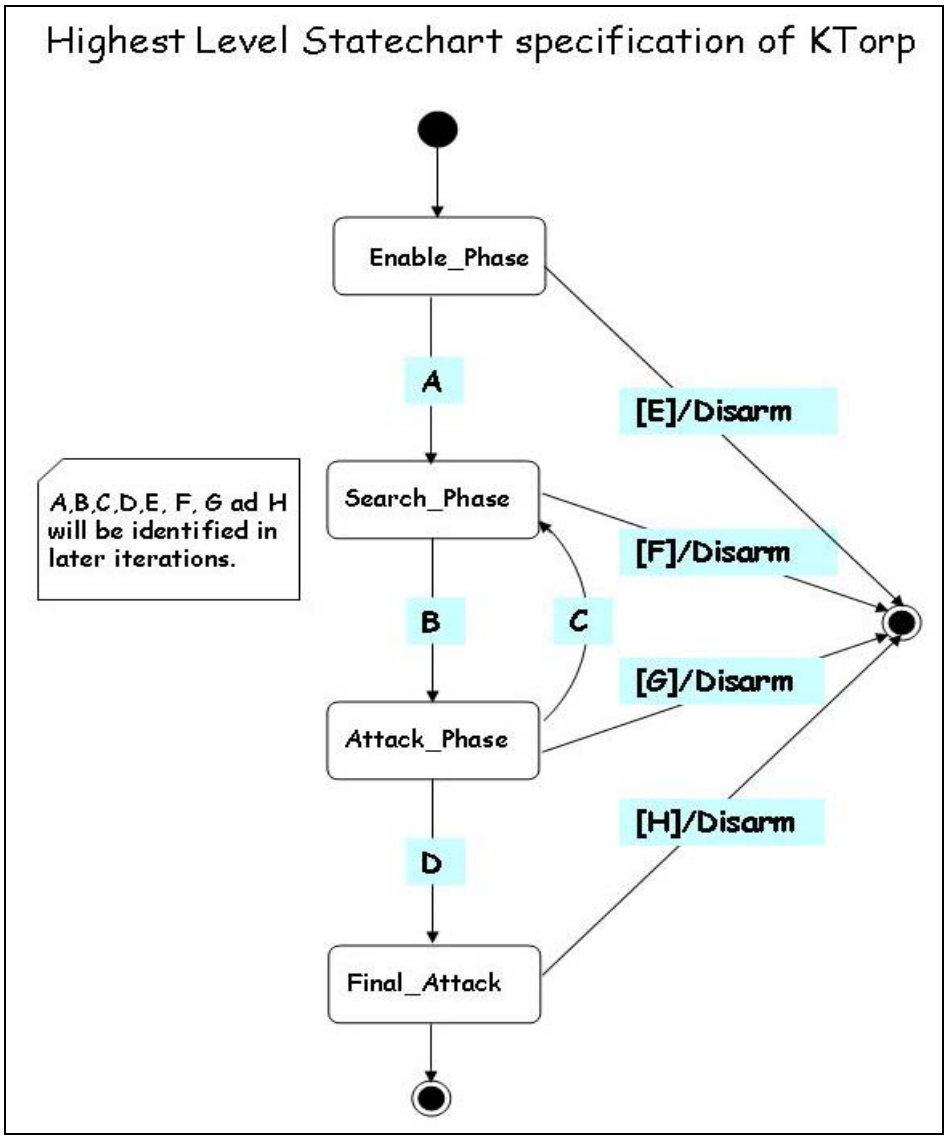


Figure IV-9. Highest Level Statechart Specification of KTorp

"Enable_Phase" state is considered a simple state and it will not be discussed in detail in this study. The specification SS.1 and SS.2 provides the information necessary for identifying the substitute "A" in Figure IV-9. For the rest of the study, such substitutions can easily be identified when transitions between states are clarified.

SS.2. When KTorp reaches the enable distance, it will **arm itself** and its **sonar** will become **active**. At this point, **KTorp enters the search phase**.

The identified transition from the "Enable_Phase" state to the "Search_Phase" state is shown in Figure IV-10.

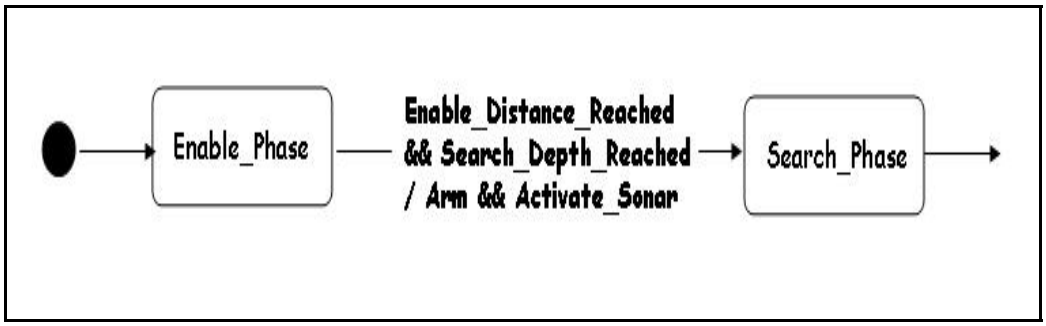


Figure IV-10. Transition between "Enable_Phase" State and "Search_Phase" State

At this point, the "Search_Phase" is analyzed. It is recognized that there are two main spanning behaviors throughout the torpedo run. The first behavior is about the navigation of KTorp. The second behavior is the reception and analysis of the signals. These behaviors are represented by simple concurrent states within composite states. For example, using the natural language specification SS.3 and SS.4, the "Search_Phase" is identified as a composite state, including two concurrent states. These states are "Signal_Reception_of_5msec_Pulse_Length" and "Snake_Navigation_Pattern_Type_I". In this study, to facilitate the understanding of the specification, long state names are used. State name "Snake_Navigation_Pattern_Type_I" denotes the behavior explained in SS.3.

The natural language specification SS.3 and SS.4 are as follows.

SS.3. In the search phase, KTorp's navigation is just like a snake. First, KTorp will steer 15° right angle ("starboard" in Navy lingo) and navigates 300 yards. Then, KTorp will steer 30° left angle ("port" in Navy lingo) and 30° right angle after navigating 300 yards in each turn. This pattern continues until KTorp enters the attack phase.

SS.4. When the torpedo receives two signals with 5 msec. pulse length within five seconds, KTorp enters the attack phase. (The noise in the environment is filtered by the sonar and these signals are certain to be generated by a potential target with a high probability)

Further refinement of the "Search_Phase" composite state is given in Figure IV-11.

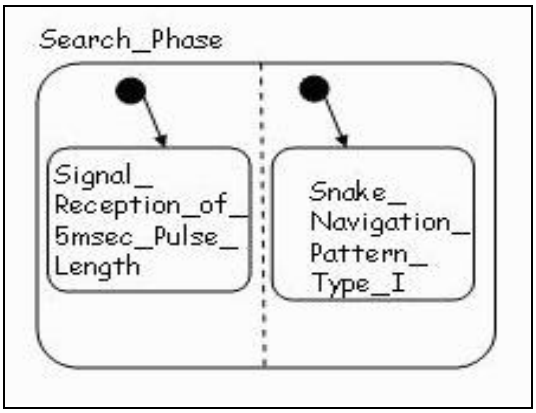


Figure IV-11. Refinement of "Enable_Phase" Composite State for Snake Search Mode

The second iteration continues with the first level refinement of the composite states. The composite states are "Search_Phase," "Attack_Phase" and "Final_Attack". The same refinement is also accomplished for the circular search mode. The composite state for "Search_Phase" in circular search mode is given in Figure IV-12.

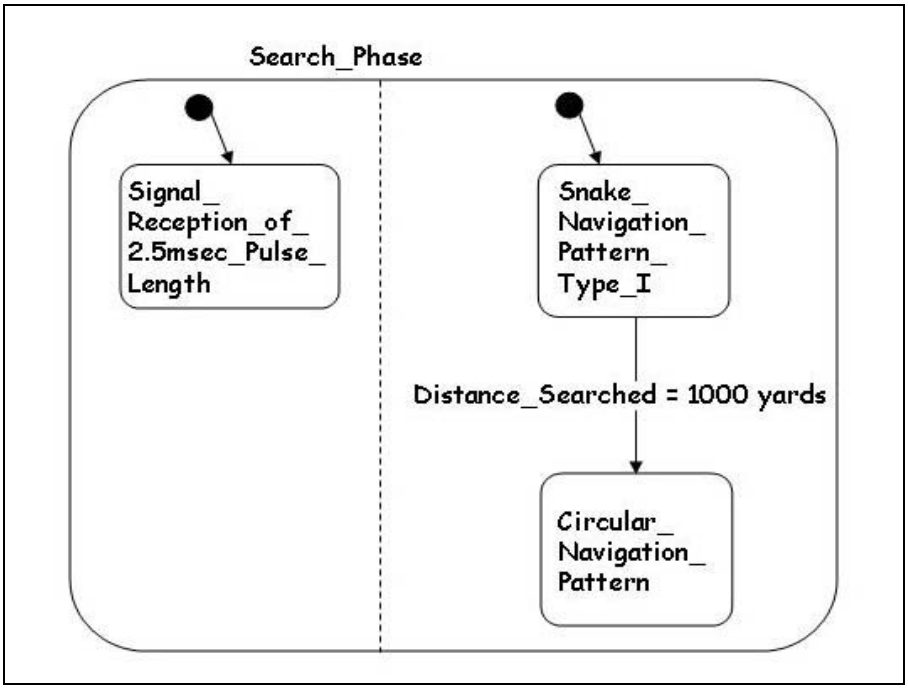


Figure IV-12. Refinement of "Enable_Phase" Composite State for Circular Search Mode

At this point of the development, the general layout of the statechart specifications for two different search modes is outlined. In the third iteration, the details of the specifications are under scope. Therefore, for each transition, events and guard

conditions are identified. If both search modes are analyzed, it will be seen that the transition from the "Enable_Phase" state to the "Search_Phase" state is the same for both modes. This is derived from the natural language specifications SS.1, SS.2, CS.1 and CS.2.

The identified guards and events are highlighted. Figure IV-13 shows the transition. Earlier in the study, it is mentioned that the transitions will be identified. In Figure IV-9, the transition A corresponds to transition "Enable_Distance_Reached && Search_Depth_Reached / Arm && Activate_Sonar" in Figure IV-13.

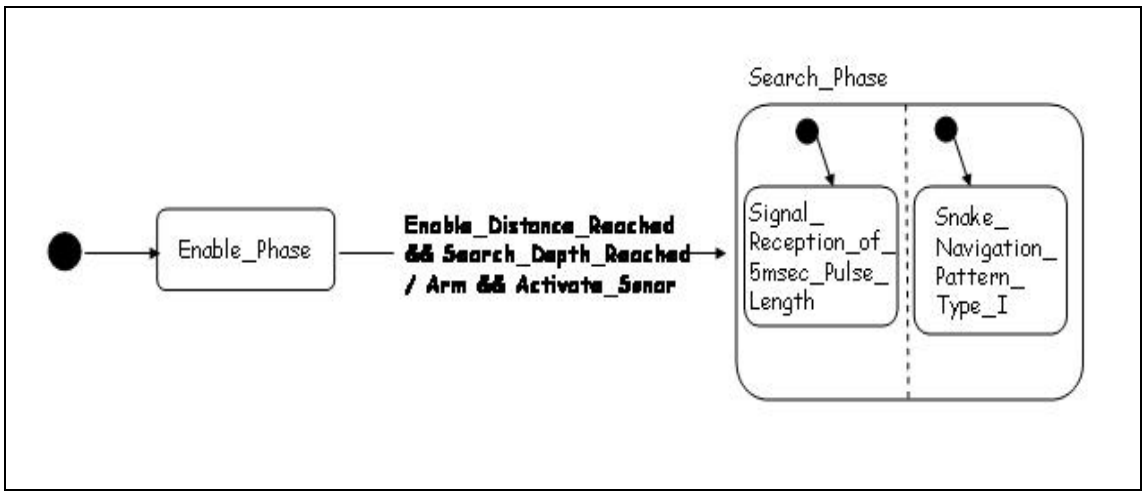


Figure IV-13. Transition between "Enable_Phase" State and "Search_Phase" State

Then, the iteration continues with analyzing snake search mode. The transitions are derived from the natural language specifications and Snake Search use case (UC-1). The highlighted portion of the specification SS.4 gives the necessary information for the transition.

SS.4. When the torpedo **receives two signals** with five msec. pulse length **within five seconds**, KTorp **enters attack phase**.

Since the pulse length for the signals are the same, it is assumed that the received signals satisfy this condition. This assumption will simplify the statechart specification at this point. Signals with other pulse lengths will not satisfy the necessary guard conditions. The transition from the "Search_Phase" state to the "Attack_Phase" state is

accomplished as shown in Figure IV-14. Note that a temporary state is added to the statechart specification.

The same approach used in the refinement of the "Search_Phase" composite state is applied to the "Attack_Phase" state. The navigation pattern described in SS.5 is called "Snake_Navigation_Pattern_Type_II".

SS.5. In the attack phase, KTorp focuses on the potential signal source. In this mode, first the torpedo steers 7.5° right angle and navigates 150 yards. Then, KTorp will steer 15° left and 15° right angle after navigating 150 yards in each turn.

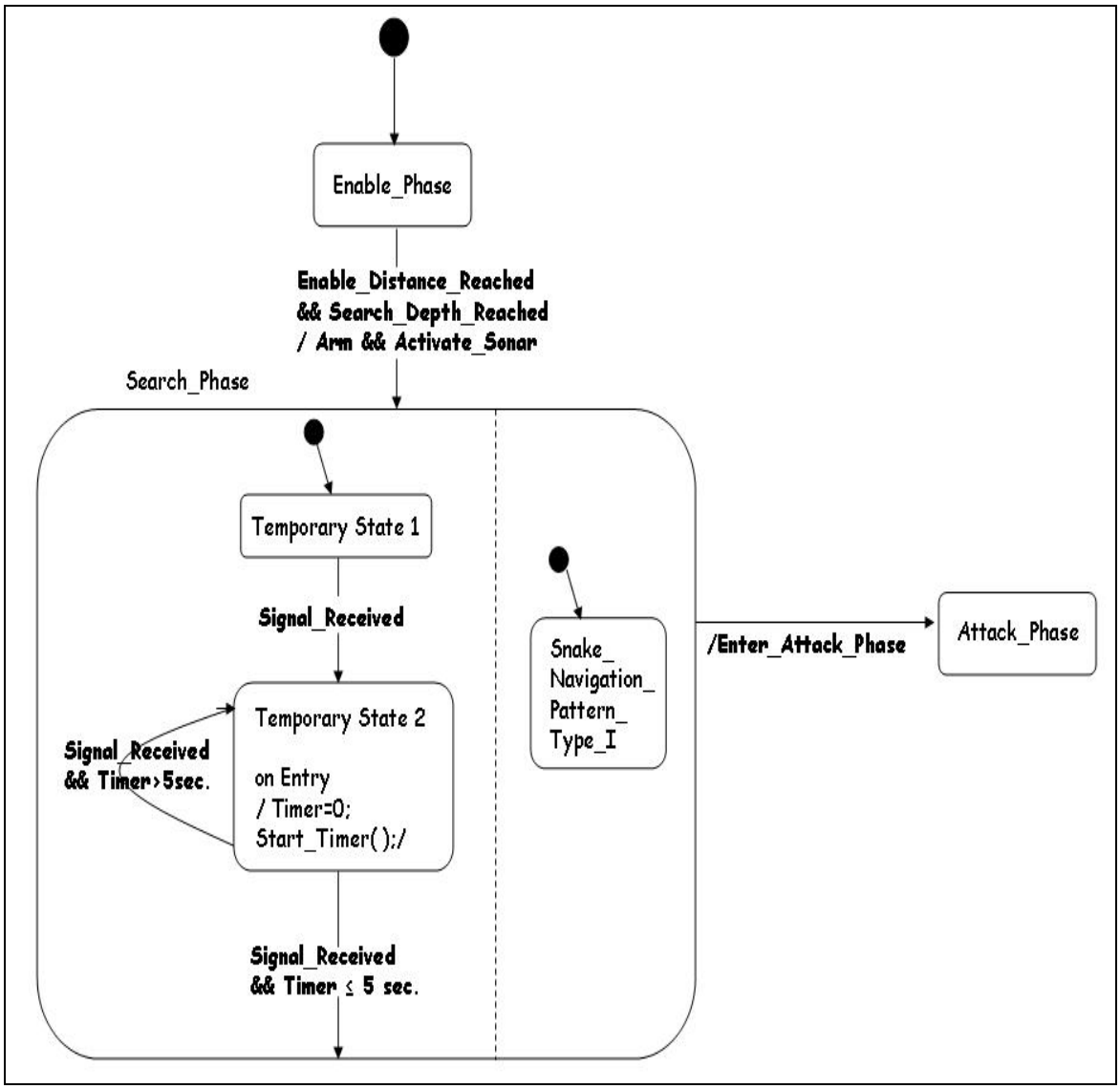


Figure IV-14. Transition from "Search_Phase" State to "Attack_Phase" State

SS.6. When the torpedo **receives three signals** with 2.5 msec. pulse length **within three seconds**, KTorp decides that the signal source is a real target and attacks the target.

Using natural language specification SS.5 and SS.6 and Snake Search use case (UC-1), the refinement of the "Attack_Phase" is achieved as shown in Figure IV-15. The behavior specified in SS.6 is called "Signal_Reception_of_2.5msec._Pulse_Length".

At this point of the development, it is observed that achieving the specification with deterministic Harel statecharts causes some problems. An attempt to specify the behavior outlined in SS.6 with deterministic Harel statecharts is shown in Figure IV-16.

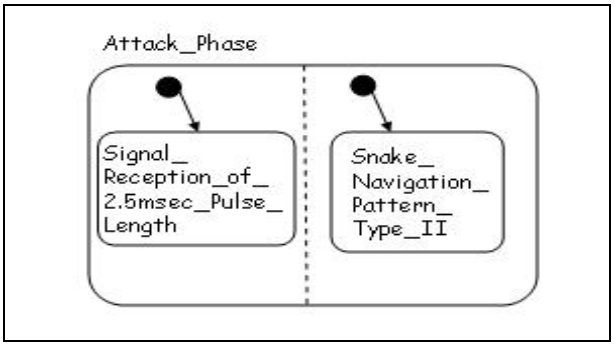


Figure IV-15. Refinement of "Attack_Phase" Composite State for Snake Search Mode

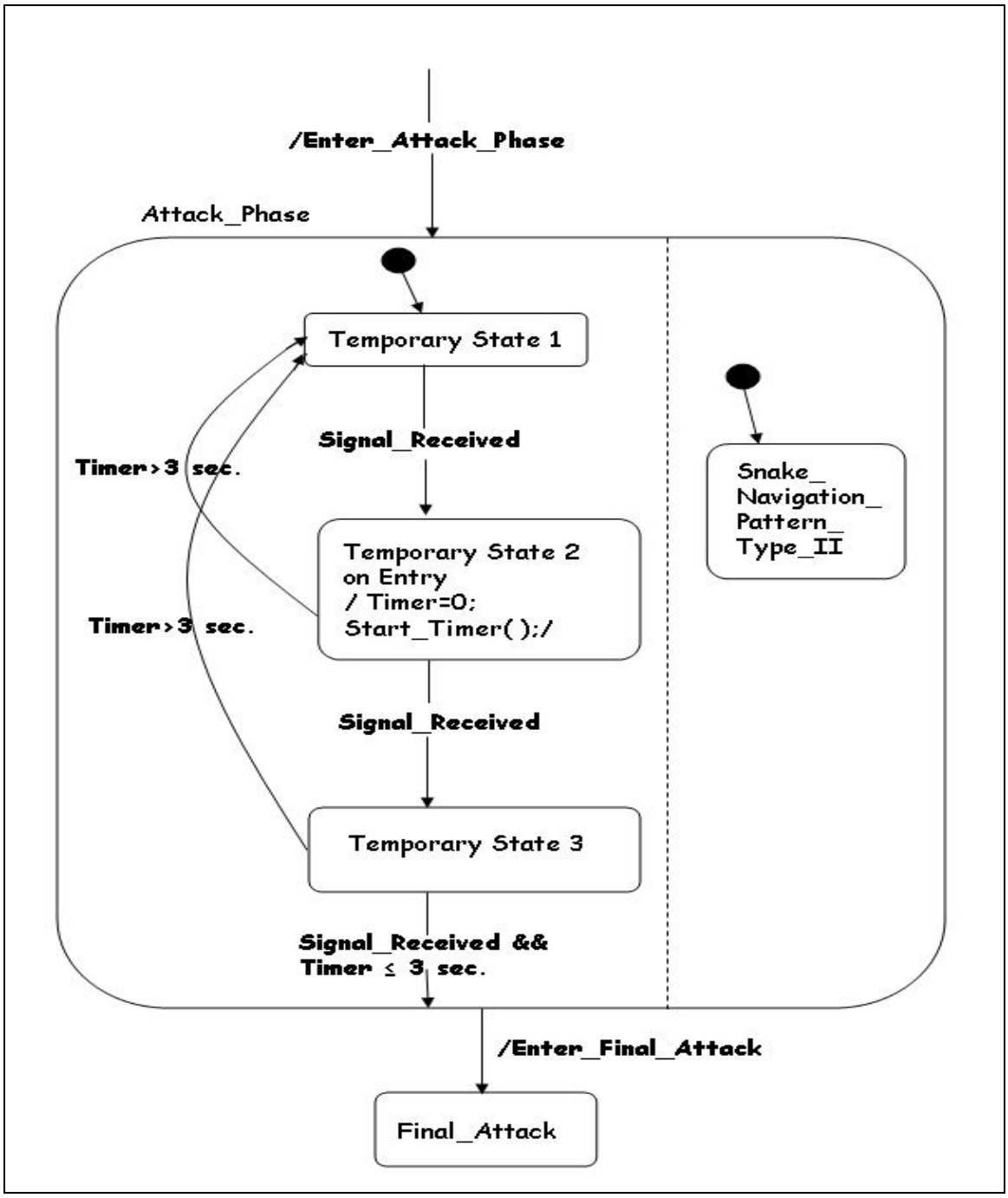


Figure IV-16. An Attempt to Achieve "Attack_Phase" Composite State for Snake Search Mode

At first, the statechart given in Figure IV-16 may seem to reflect the behavior required. However, when carefully analyzed, it will suffer the problem depicted in Figure IV-17.

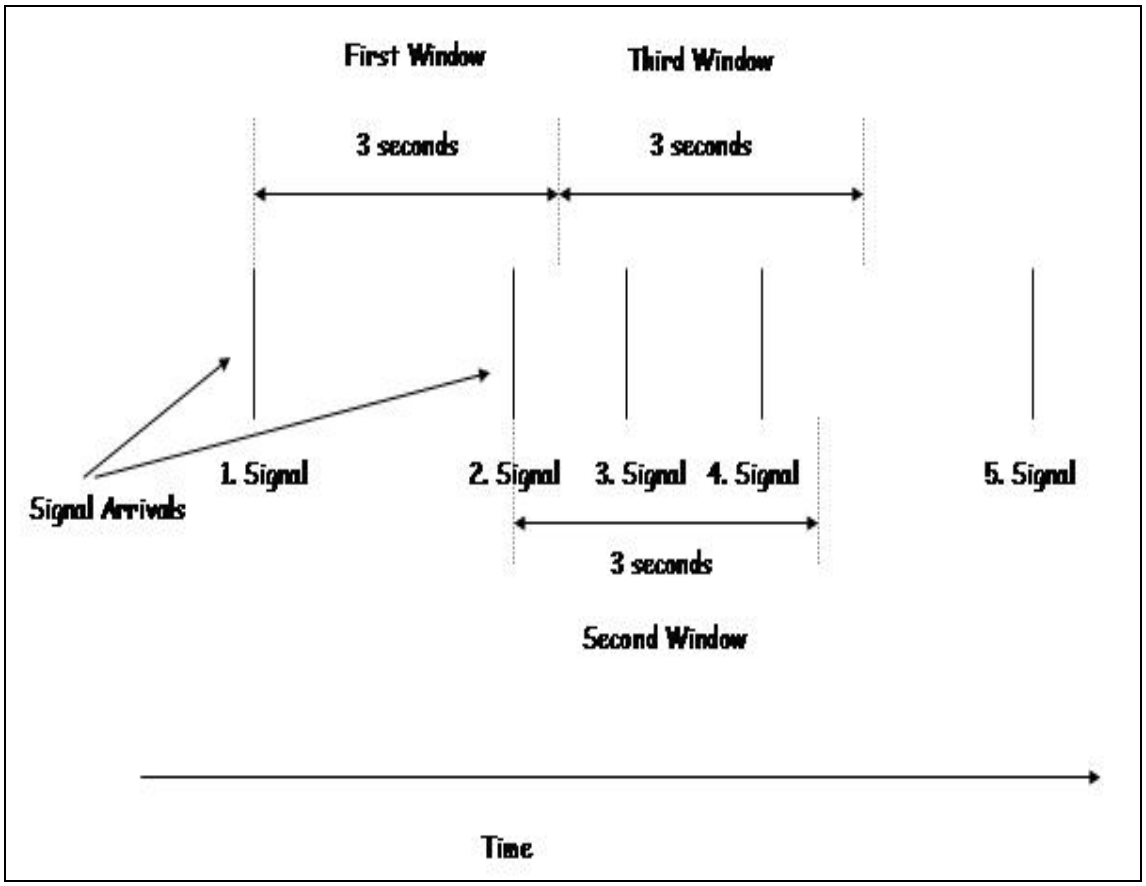


Figure IV-17. Sliding Window Problem

Figure IV-17 represents a simple scenario for the attack phase. The signals are arriving randomly. The problem is that even though the arriving pattern of signals satisfies the necessary condition, the deterministic Harel statechart given in Figure IV-16 will not recognize the pattern. First, the implementation will look into the first window and return to temporary state 1. Then, the implementation will look into the third window. Still, the pattern will be seen as unsatisfactory for the condition. However, when a second window is captured from the pattern, it will be observed that this window satisfies the condition. This window will not be recognized by the deterministic Harel statechart given in Figure IV-16.

Figure IV-18 shows a deterministic Harel statechart specification for the requirement SS.6. In this statechart, two timers are used to capture the second window depicted in Figure IV-17.

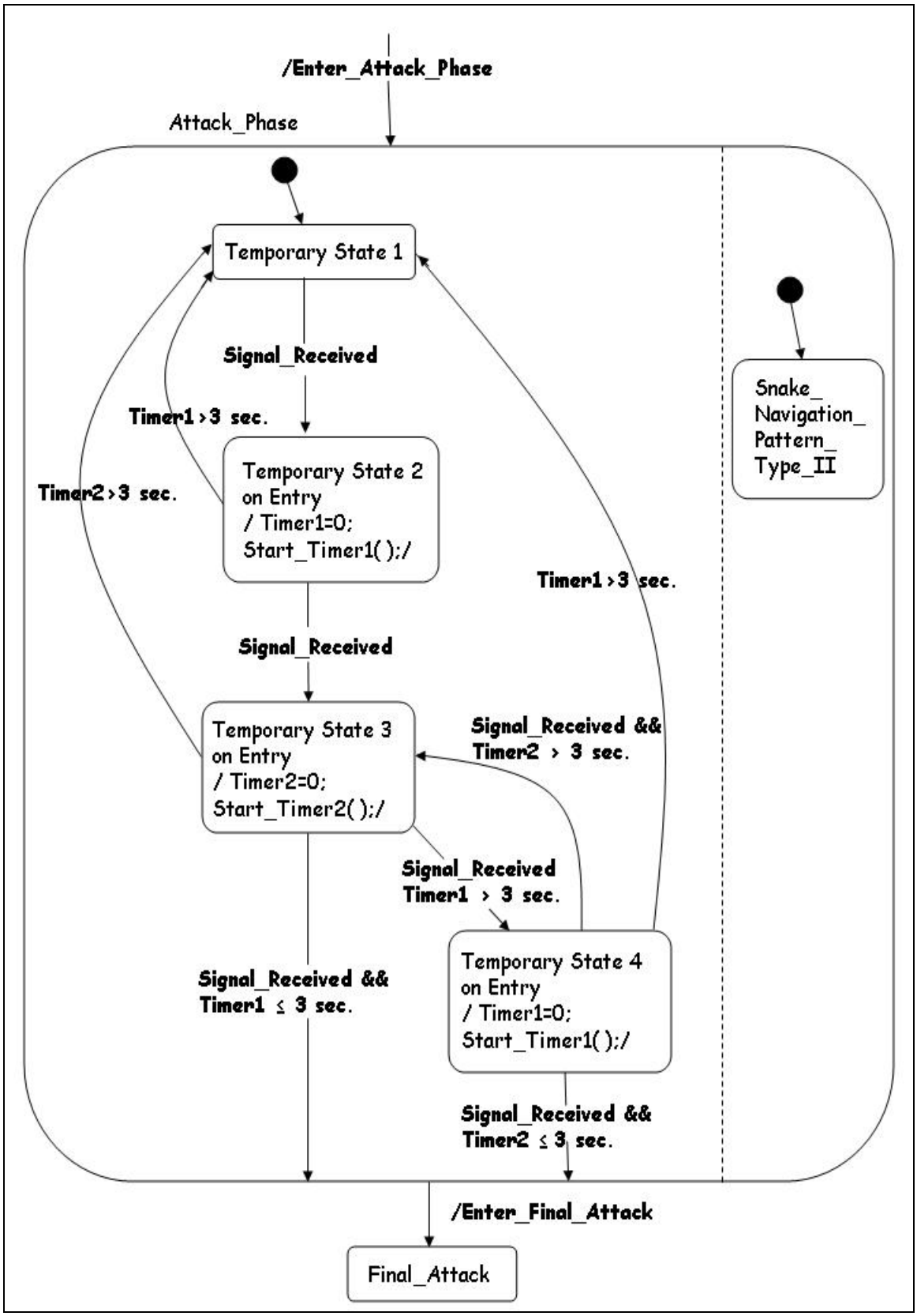


Figure IV-18. Deterministic Harel Statechart Specification for the Requirement SS.6

The deterministic Harel statechart given in Figure IV-18 uses two different timers to keep track of the sliding windows shown in Figure IV-17. If the requirement SS.6 required more than 3 signals, the deterministic Harel statechart would have been more complex. The number of the timers and the number of windows that needs to be observed will increase. Such requirements necessitate more states and complex specifications with deterministic Harel statecharts.

At this point, a non-deterministic approach will be used to specify the requirement given with SS.6. Non-determinism will overcome the difficulties mentioned as the sliding window problem. A simple TLChart will capture the requirement. Figure IV-19 shows the assertion statechart for the specification of the requirement. Later, this assertion statechart will be added to the primary statechart. This assertion is called "Assertion1".

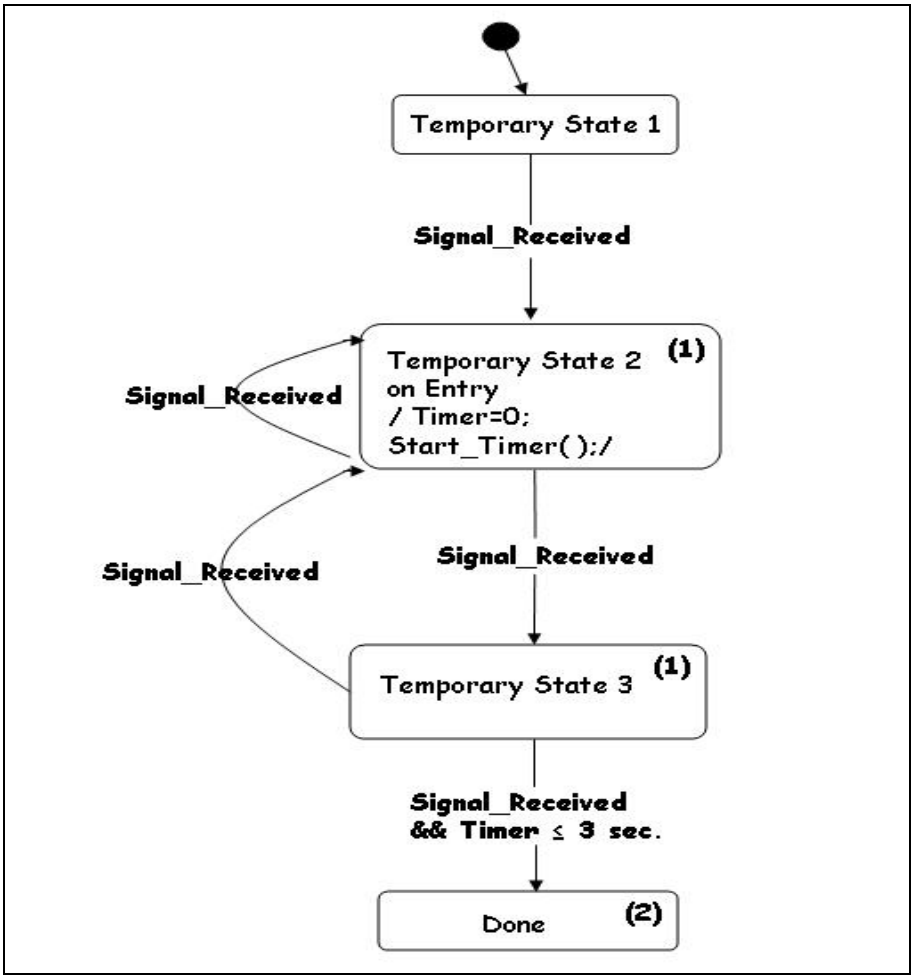


Figure IV-19. Non-deterministic Solution for Requirement SS.6

In Figure IV-19, there are two transitions labeled “Signal_Received” leaving from temporary state 2. This is how non-determinism is used. Such transitions may create ambiguity during execution on a deterministic computer. TLCharts overcome the ambiguity using a priority scheme. Transitions that lead to a good state (in the figure, the good state is the state Done) have a higher priority during execution. Other transitions with the same condition have lesser priority. Priorities are shown in the upper corner of the states for the sake of clarity. Normally, these priorities are not added to the specification. They are handled internally by the tool used to process the TLChart. Temporary state 2 and 3 have lesser priority than Done state. The smaller the number is, the less priority the computation has. When priorities are the same, a random selection can be made or however the implementation of the tool enforces.

The natural language requirement SS.7 also needs to be added to the specification. This requirement adds a time limit for the attack search phase, which is defined in requirement SS.6. The requirement SS.7 is as follows.

SS.7. If KTorp **does not receive the signals** specified in step 6 **within one minute**, it **returns to the search phase**.

The non-deterministic statechart shown in Figure IV-19 is embedded in the overall specification as an assertion statechart. Figure IV-20 shows the primary statechart specification.

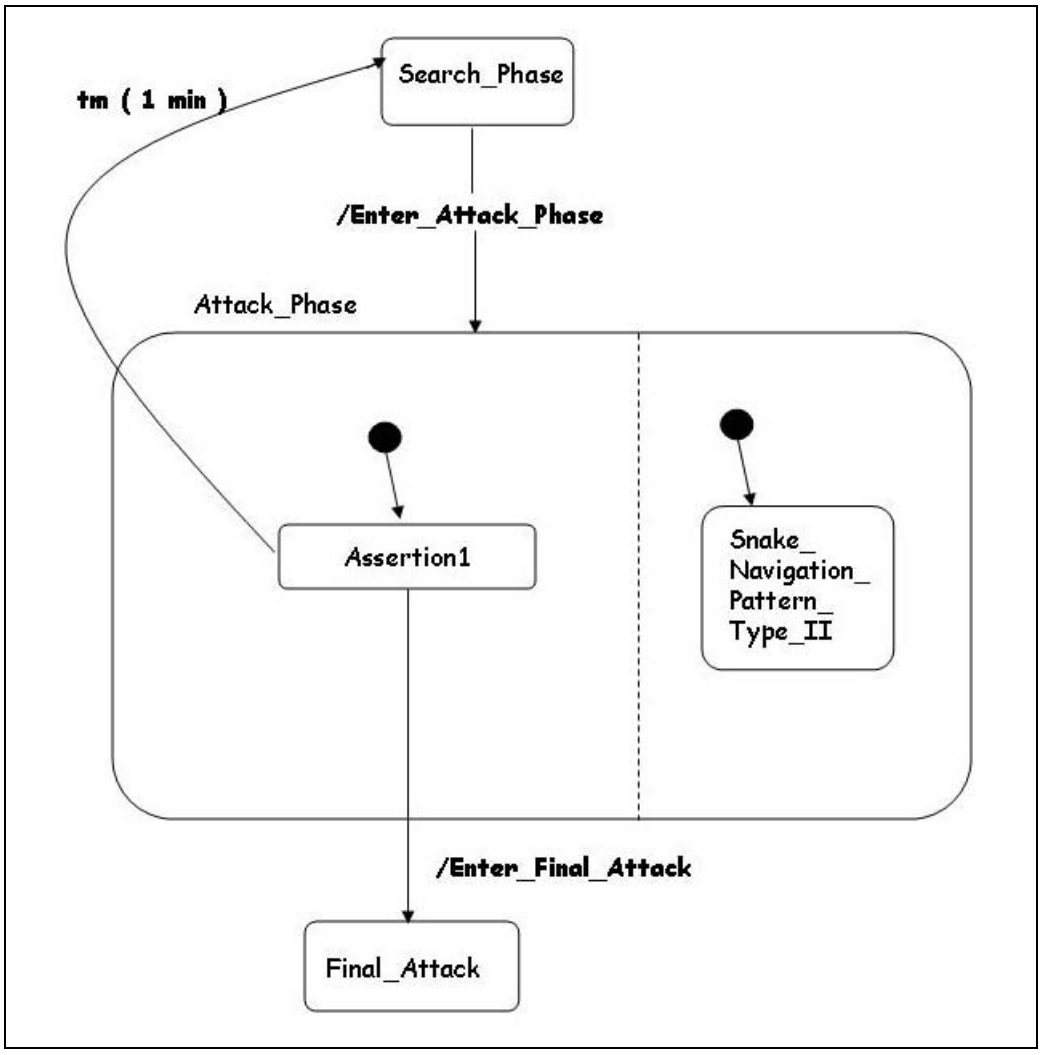


Figure IV-20. Primary Statechart Specification for Attack Phase

Adding such requirements as assertions to the overall statechart specifications has some advantages. These advantages will be discussed later.

The Final Attack phase is described with requirement SS.8. The requirement is as follows.

SS.8. While KTorp is attacking the target, it follows a straight path until it destroys the target or the torpedo run ends.

The final attack phase will be specified with only one simple state shown in Figure IV-20.

At this point, the specification process is finalized for the purpose of this study. The goal of this study was to show that the use of TLCharts when using deterministic Harel statecharts is inefficient. Non-determinism is an important part of TLCharts. Every requirement that can be specified with non-deterministic statecharts, can also be specified with deterministic Harel statecharts. However, the number of states increases exponentially. The development of such statecharts will be costly. Using non-determinism generally provides an efficient specification when compared to not using it. TLCharts also enables the use of temporal logic conditioned transitions within the specification. Drusinsky [37] states that

> TLCharts visually and intuitively resemble Harel statecharts while enabling temporal-logic conditioned transitions. This is useful for the specifying of non-deterministic temporal properties inside a statechart specification, thereby combining the simple and familiar statechart notation with temporal logic ability to capture negation and non-determinism.

As a result, the final statechart specifications will arguably be easier to understand, read, and maintain.

In this study, KTorp is used as the example. KTorp is a close example of real torpedoes. While developing KTorp statecharts, it is shown that using deterministic Harel statecharts may cause some problems mentioned as the sliding window problem. TLCharts with non-determinism solves the problems as shown in Figure IV-20.

I. ASSERTIONS

An assertion is generally defined as a programming language construct that checks whether an expression is true. For many years, programmers used assertions in order to simplify debugging throughout development. Most widely-used programming languages have mechanisms for assertions. The use of assertions is more widespread now. Today, assertions are also used for specification verification purposes.

Formal specification assertions can be used with Run-time Execution Monitoring (REM) in order to track the temporal behavior of an underlying application. REM methods range from printing messages to run-time tracking of complex formal requirements for verification purposes. Such methods have recently been used by NASA for the Deep Impact project [38].

Published REM methods generally use temporal logic as a specification language. Conventionally, run-time verification methods have been used in the later stages of the design to validate and debug the code. Correctness assertions are written and used for REM-based testing or for model checking. These are called test-time assertions. Drusinsky, Shing and Demir recently presented two more additional assertion types [3]:

- Assertions that are used only during simulation
- Deployable assertions integrated with the run-time control flow of the target software

Test-time assertions are used for testing the correctness of the design or the implementation or both. Assertions that are used only during simulation are called simulation-time assertions. These assertions use the information about the environment that is not present in run-time. Deployable assertions integrated with the run-time control flow of the target software are called run-time assertions. These assertions are embedded in the application. The assertion will check whether a requirement is violated in run-time. When the requirement is violated, the assertion affects the control flow of the software via an exception handling routine.

A deployable assertion example is given in Figure IV-21. The figure shows a TLChart version of the statechart shown in Figure IV-9, where the additional transition from Attack_Phase to Search_Phase will force the torpedo control software to switch back to Search_Phase if the temporal proposition (□>$_{1\ min}$ **(in Final_Attack)**) evaluates true.

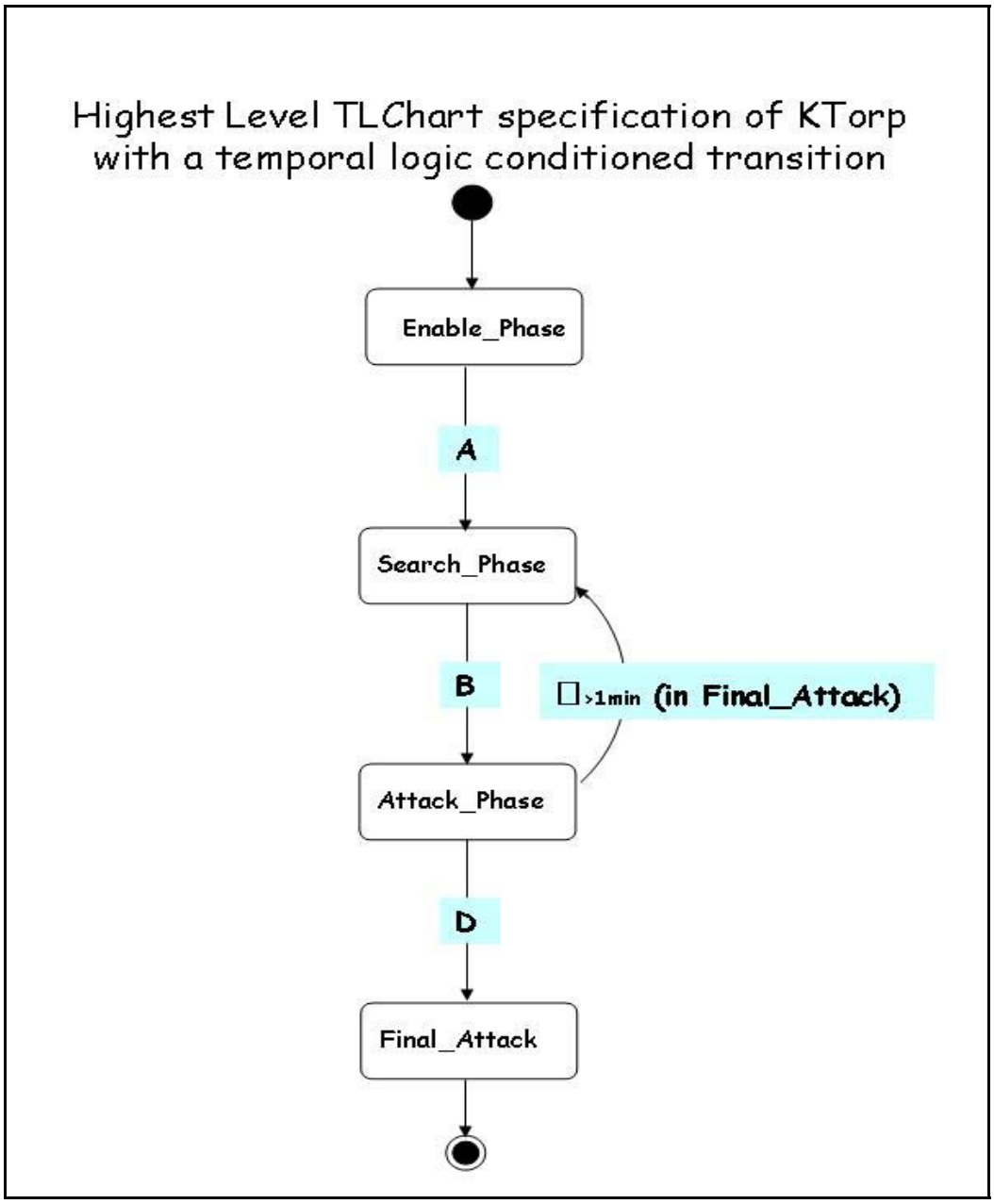


Figure IV-21. Highest Level TLChart Specification of KTorp with a Temporal Logic Conditioned Transition

Note that the temporal logic conditioned transition corresponds to substitute C in Figure IV-9.

Further details about assertion types with examples can be found in [3].

In the development of KTorp specification, the requirement SS.6 is integrated into the primary statechart as an assertion statechart. This assertion could also be written in

temporal logic. However, the belief is that using the same specification language for both modeling the system and verification of the specification will be beneficial to the developers.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSION AND FUTURE WORK

A. SUMMARY

The scope of this thesis was to investigate the suitability of TLCharts, a specification language that combines statecharts and temporal logic, for the early specification of dynamic characteristics of a homing torpedo. The homing torpedo example, KTorp, made it possible to analyze TLCharts in a weapons systems software context. TLCharts is a recently-proposed specification language. Therefore, the belief was that such analysis will help in understanding the advantages and/or disadvantages of using TLCharts for the early specification of systems. In order to achieve this goal, KTorp was specified using a systematic software development process. First, KTorp was conceived and defined using natural language specifications. The development of use cases and system sequence diagrams followed natural language specifications. The early specification process was finalized with statechart specifications.

Software development starts in the minds of developers. Then, the ideas turn into natural language specifications. The process follows with early specifications and continues further. Every transformation of information increases the possibility of error. Therefore, some form of verification is required after every transformation. Barry Boehm pointed out that the cost of fixing an error occurred in early stages increases as the project progresses. Every day, computer-aided development tools are getting better and new tools are being developed. Complex specifications in temporal logic, in statecharts, or in other specification languages can be analyzed and evaluated with these tools. However, it should not be forgotten that the process, especially early stages, still heavily depends on human intervention. Thus, the specification language used for early stages must be intuitive, logical, easy to understand, write and modify. TLCharts is an attempt to answer such questions.

TLCharts is a visual specification language that combines the visual and intuitive appeal of non-deterministic Harel statecharts with formal specifications written in Linear-time Temporal Logic (LTL) [37]. Harel statecharts are visual and deterministic. Linear-time temporal logic is textual/logical and non-deterministic. TLCharts combine both

formalisms which makes it possible to create specifications that are visual, partially deterministic and at the same time logical and non-deterministic. Both highly nested temporal logic conditions in a single state specification and a fully-deterministic, implementation level detailed Harel statechart specification are legal TLCharts. However, such use only abuses the purpose of TLCharts. [37]

In the KTorp example, the statechart specification started with deterministic Harel statecharts, which is in fact a legal TLChart. During the specification process, it is observed that to specify some system requirements with fully deterministic Harel statecharts may be inefficient. One example is given and explained as the sliding window problem. The requirements in the form of some events within a limited amount of time cannot easily be specified and verified with deterministic Harel statecharts. A non-deterministic solution is simpler and easier to understand. If Figures IV-16, IV-18 and IV-19 are compared, it will be observed that Figure IV-19 has fever states and transitions. Also, Figure IV-19 requires less time to understand. The author spent a considerable amount of time to derive with a deterministic solution. This experience demonstrated when some of the requirements are heavily dependent on time, and that specification and verification of those requirements with TLCharts are easier to develop.

My belief is that TLCharts is a powerful tool for early specification. The specification language has powerful features such as visual appeal of statecharts, non-determinism, and enabling temporal-logic conditioned transitions. It even allows us to use natural language conditioned transitions [37]. Drusinsky and Shing [36] provided an example on using TLCharts for armor-plating specifications. I agree with Drusinsky's warning: "Clearly, TLCharts can be abused." [37]. While using the features of the formalism, the developers should not forget the goal, which is creating clear, easily understandable and maintainable specifications. The features should be used when necessary.

Weapons systems software development is an expensive process. It takes a long time and considerable effort throughout the life cycle of the system. The major part of global software production occurs in the United States. A significant portion of this production is sponsored by the Department of Defense. Numerous reports indicate that

most of the projects could not be completed or used effectively due to various reasons. Every effort to improve software production will be extremely beneficial. The study aims to contribute with an analysis of newly-proposed specification language in a special context. As a result, this thesis demonstrated that using TLCharts as the early specification language for weapon systems software provides efficient, visual and intuitive specifications.

B. FUTURE WORK

TLCharts also makes it possible to use temporal-logic conditioned transitions. However, in this thesis, the KTorp example did not require such use. There are two reasons. First, the example was simplified and did not include complex requirements. Second, including examples with temporal-logic conditioned transitions will unnecessarily broaden the subject for one thesis study. Also, the KTorp example with its current form is hard to understand for unfamiliar readers. A follow-up study may be the analysis of "just in time temporal logic" property of TLCharts.

Safety requirements are of utmost importance for weapon systems software. These safety requirements are generally in the form of what the system must not do. TLCharts provides an opportunity for armor-plating specifications. It is achieved via over-specification of a fully specified design with temporal logic conditions to strengthen the safety of the system. Such a concept is introduced with an example in [36]. A follow-up study may answer the questions of when and how much armor-plating is useful.

The formalism should be supported with the necessary tools. Otherwise, it will only stay as another proposed formalism in the literature. After the necessary tools are in place, the software developers will have a chance to apply TLCharts in their projects. The true evaluation of applying TLCharts will only be revealed after experience in real projects. Currently, there are two separate tools called DBRover and Temporal Rover developed by Time-Rover Inc [38]. These tools may be analyzed and tested with an example to reveal their capabilities to support features of TLCharts.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] D. Drusinsky, "Semantics and Runtime Monitoring of TLCharts: Statechart Automata with Temporal Logic Conditioned Transitions," *Proc. 4th Runtime Verification Workshop (RV'04)*, 2004, Invited paper.

[2] Harel, D. "Statecharts: A Visual Formalism for Complex Systems," *Sci. Comput. Program. 8, 3*, June 1987, pp. 231-274.

[3] D. Drusinsky, M. Shing, and K. Demir, "Test-time Run-time, Simulation-time Temporal Assertions in RSP," *Proc. 16th IEEE International Workshop on Rapid System Prototyping,* Montreal, Canada, 8-10 June, pp. 105-110.

[4] Stephen R. Schach, *Object-Oriented and Classical Software Engineering, Fifth Edition*, p. 30, McGraw Hill, 2002.

[5] E. Yourdon, *Rise and Resurrection of the American Programmer,* Yourdon Press, Upper Saddle River, NJ, 1996 (Chapter 1).

[6] T. Drake, "Testing Software Based Systems: The Final Frontier," Software Tech News, Volume 3 Number 3 - Software Testing Part 2.

[7] "Report of the Defense Science Board Task Force on Military Software," Office of the Under Secretary of Defense for Acquisition, Washington, DC, September 1987. (Chapter 2).

[8] M. C. Paulk, B. Curtis, M. B. Chrissis, C.V. Weber, "Capability Maturity Model for Software, Version 1.1," Technical Report CMU/SEI-93-TR-024 ESC-TR-93-177, February 1993.

[9] W. W. Royce, "Managing the Development of Large Software Systems," *Proceedings,* IEEE, August 1970.

[10] B. W. Boehm, "A Spiral Model of Software Development and Enhancement," IEEE Computer, May 1988, pp. 30-44.

[11] "Report of the Defense Science Board Task Force on Defense Software," November 2000, Defense Science Board, pp. 11.

[12] Defense Science Board Official Web Site (http://www.acq.osd.mil/dsb/), November 2005.

[13] "Report of the Defense Science Board Task Force on Military Software," September 1987, Defense Science Board.

[14] Beam, W., Chairman, Air Force Studies Board, *Adapting Software Development Policies to Modern Technology*, National Academy Press, Washington, D.C., 1989.

[15] Crosstalk – The Journal of Defense Software Engineering Web Site (http://www.stsc.hill.af.mil/crosstalk/about.html), November 2005.

[16] Hassan Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*, p. 8, Addison-Wesley, 2000.

[17] Harel, D., and M. Politi. *Modeling Reactive Systems with Statecharts.* New York, McGraw Hill, 1998.

[18] J. Voas, W. W. Agresti, "Software Quality from a Behavioral Perspective," *IT Pro,* IEEE Computer Society, August 2004, pp. 46-50.

[19] Capers Jones, "Defense Software Development in Evolution," *Crosstalk-The Journal of Defense Software Engineering,* November 2000.

[20] CHAOS Study, Standish Group, 1999.

[21] Von Der Beek, M., "A Comparison of Statechart Variants," In *Formal Techniques in Real-Time and Fault-Tolerant Systems,* L. de Roever and J. Vytopil, Eds. Lecture Notes in Computer Science, vol. 863. Spriger-Verlag, New York, pp. 128-148.

[22] Object Management Group, *OMG Unified Modeling Language Specification*, March 2003, Version 1.5, Part 9.

[23] D. Harel, A. Naamad, "The STATEMATE Semanics of Statecharts," *ACM Transactions on Software Engineering and Methodology,* Vol. 5, No. 4, October 1996, pp. 293-333.

[24] M. Fisher, "Temporal Logic: Introduction," COMP313 / COMP513 Course Notes, University of Liverpool.

[25] A. Pnueli, "The Temporal Logic of Programs," *Proc. 18th IEEE Symp. on Foundations of Computer Science*, 1977, pp. 46-57.

[26] A. Pnueli, "Applications of Temporal Logic to the specification and Verification of Reactive Systems: A Survey of Current Trends," *Lecture Notes in Computer Science #224,* 1985, pp. 510-584.

[27] P. Bellini, R. Mattolini, P. Nesi, "Temporal Logics for Real-Time Specification," *ACM Computing Surveys,* Vol. 32, No. 1, March 2000.

[28] National Institute of Standards and Technology Official Website: (http://www.nist.gov/dads/HTML/temporllogic.html), November 2005.

[29] Prior, A *Past, Present, and Future.* Oxford University Press, Oxford, UK, 1967.

[30] A. Pnueli, "The Temporal Semantics of Concurrent Programs," *Theor. Comput. Sci. 13*.

[31] Clarke, E. M., Emerson, E. A., and Sistla, A. P., "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Trans. Program. Lang. Syst. 8,* April 1986, pp. 244-263.

[32] Koymans, R., "Specifying Real-Time Properties with Metric Temporal Logic," *Real-Time Syst. 2,* 4, November 1990, pp. 255-299.

[33] Ben-Ari, M., Pnueli, A., and Manna, Z. "The Temporal Logic of Branching Time," *Acta Inf. 20,* 1983.

[34] J. Hopcroft, R. Motwani, and J. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison Wesley, 2nd Edition, 2001, ISBN 0-201-44124-1.

[35] D. Drusinsky, "Monitoring Temporal Rules Combined with Time Series," *Proc. 2003 Computer Aided Verification Conference (CAV)*, pp. 114-117.

[36] D. Drusinsky and M. Shing, "TLCharts: Armor-plating Harel Statecharts with Temporal Logic Conditions," *Proc. 15th IEEE International Workshop in Rapid Systems Prototyping*, 28-30 June 2004, pp. 29-36.

[37] D. Drusinsky, "Visual Formal Specification using (N)TLCharts: Statechart Automate with Temporal Logic ad Natural Language Conditioned Transitions," *Parallel and Distributed Systems: Testing and Debugging Workshop 2004,* April 30, 2004, Santa Fe, New Mexico.

[38] http://www.time-rover.com , November, 2005.

[39] J.M. Wing, "A specifier's introduction to formal methods," Computer, Volume 23, Issue 9, September 1990 Page(s): 8-22, 24.

[40] V. Berzins, L. Luqi, *Software Engineering with Abstractions*, Addison Wesley, 1991, ISBN 0-201-08004-4.

[41] S. Bennett, J. Skelton, K. Lunn, *UML, Schaum's Outline Series*, McGraw-Hill, 2001, ISBN 0-07-709673-8.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Doron Drusinsky
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California

4. Man-Tak Shing
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California

5. Deniz Kuvvetleri Komutanligi (Turkish Navy Headquarters)
   Ankara, Turkey

6. Deniz Harp Okulu Komutanligi (Turkish Naval Academy)
   Istanbul, Turkey

7. Arastirma Merkezi Komutanligi
   Yazilim Gelistirme Grup Baskanligi
   (Turkish Navy Software Development Center)
   Istanbul, Turkey

8. Deniz Bilimleri Enstitusu ve Muhendisligi Mudurlugu
   (Turkish Naval Science Institution and Engineering School)
   Istanbul, Turkey